# Realisierungsmethodik von applikationsspezifischen Softcore FPGA-Lösungen

in Abhängigkeit von algorithmischen Anforderungen im Einsatzgebiet eingebetteter Systeme

Dissertation
zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)

vorgelegt der
Fakultät für Informatik und Automatisierung
der Technischen Universität Ilmenau

von
Master of Science Michael Kirchhoff
geboren am 25.10.1989 in Nordhausen

vorgelegt am: 10.01.2019
Tag der wissenschaftlichen Aussprache: 12.11.2019

1. Gutachter: Univ.-Prof. Dr.-Ing. habil. Wolfgang Fengler
2. Gutachter: Univ.-Prof. Dr.-Ing. habil. Andreas Mitschele-Thiel
3. Gutachter: Prof. Dr. Eng. Mihai Cernat

*urn:nbn:de:gbv:ilm1-2019000709*

# Danksagung

An dieser Stelle möchte ich als Autor meinen Dank aussprechen.
Zuerst danke ich Professor Wolfgang Fengler für die Möglichkeit, am Fachgebiet Rechnerarchitektur und Eingebettete Systeme der Technischen Universität Ilmenau tätig sein und promovieren zu können.

Des Weiteren bedanke ich mich bei meinen Kollegen Dr. Bernd Däne, Dr. Detlef Streitferdt und Dr. Marcus Müller für die ausgezeichnete Zusammenarbeit im Fachgebiet und viele fruchtbare Diskussionen sowie den Beistand und die Unterstützung für dieses Vorhaben.

Weiterhin seien mit Lothar Wagner, Philipp Kerling, Jörn Weisensee, Björn-Philipp Kreuzkamp, Daniel Ölschlegel, Mikhail Votyakov und Natalia Kaptsova die Menschen gewürdigt, die ich während ihrer Studienzeit bei Projekt- und Abschlussarbeiten im Rahmen meiner Promotionstätigkeit betreut habe und die ihren Beitrag zu den Ergebnissen dieser Arbeit oder den Projekten des Fachgebiets geleistet haben.

Zuletzt möchte ich meinen persönlichen Dank aussprechen - meiner gesamten Familie und vor allem meiner Frau Valery, die mich im Verlauf der Arbeit moralisch unterstützt haben.

# Kurzzusammenfassung

Die vorliegende Dissertation befasst sich mit Prinzipien, Methodiken, Techniken und Realisierungen zur systematischen Entwicklung von komplexen eingebetteten Systemen unter Verwendung von Softcore Prozessoren. Die adressierte Aufgabendomäne ist vor allem die echtzeitkritische Daten- und Bildverarbeitung. Notwendig sind neue Lösungen aufgrund immer leistungsfähigerer eingebetteter Systeme, mit deren Hilfe Aufgabenfelder bedient werden können, die bisher mit diesen Systemen nicht umsetzbar waren.
Aufbauend auf den Darstellungen bereits existierender Modelle und Verfahren, wie z. B. dem V-Modell oder dem Hardware-Software Co-Design, wird eine spezielle Realisierungsmethodik für applikationsspezifische Softcore FPGA-Lösungen in Abhängigkeit von algorithmischen Anforderungen in der Aufgabendomäne erarbeitet. In diesem Zusammenhang wird eine Softcore-Bibliothek mit an diese Domäne angepassten Eigenschaften konzipiert und umgesetzt.
Das dabei verwendete modellbasierte Vorgehen ermöglicht durch eine hierarchische Beschreibung und Validierung eine zeit- und kosteneffiziente Entwicklung komplexer Systeme. Für jede Abstraktionsebene werden Modelle vorgestellt, die jeweils auf dieser alle notwendigen Anforderungen zur frühzeitigen Fehlererkennung und Fehlervermeidung sowie eine automatisierte Codegenerierung und Optimierungen sinnvoll umsetzen. Durch gezielte Festlegung einzuhaltender Kriterien und Entwicklungsschritte wird dabei in jeder Komponente der Toolchain eine bestmögliche Kombination von zeit- und kosteneffizienter Entwicklung mit der Sicherstellung der Einhaltung harter Echtzeiteigenschaften sowie einer Maximierung der Wiederverwendbarkeit, erreicht.
Dabei spielt die Anpassbarkeit der eingebetteten Systeme mit Hilfe von partieller Rekonfiguration, mit der das dynamische Austauschen von Teilen des Softcores oder sogar ganzer Softcore Prozessoren zur Laufzeit ermöglicht wird, eine wichtige Rolle. Es erfolgen ein praktischer Nachweis der Funktionalität der erarbeiteten Modelle sowie ausführliche Experimente über die zeitlichen Anforderungen bei der partiellen Rekonfiguration von Softcore Prozessoren. Die praktischen Ergebnisse der Arbeit zeigen deutlich die Effizienz der Entwicklung von Lösungen mit der konzipierten und umgesetzten Toolchain sowie die Relevanz und Einsetzbarkeit der partiellen Rekonfiguration in diesem Gebiet.

# Abstract

This dissertation focuses on principles, methods, techniques and realizations for the systematic development of complex embedded systems using softcore processors. The addressed domain is primarily real-time-critical data and image processing. New solutions are needed due to the increasing performance of embedded systems, allowing for a range of applications that were previously not solvable with these systems.
Building on the concepts of already existing models and methods, e.g. the V-model or hardware-software-co-design, a special realization methodology for application-specific softcore FPGA solutions is developed, in conjunction with algorithmic requirements in the addressed domain. In this context, a softcore library with characteristics tailored to this domain is designed and implemented.
Through a hierarchical description and validation, the model-based approach used in this thesis enables the time- and cost-efficient development of complex systems. For each abstraction level, models are presented that provide all necessary requisites for early error detection and prevention, as well as mostly automated code generation and code optimization. By defining relevant criteria and development steps, a parsimonious development with respect to time and cost is achieved in each component of the toolchain. This ensures strict adherence to the hard real-time properties and maximizes the reusability of the modules implemented for a specific project.
The adaptability of the embedded systems through using partial reconfiguration plays an important role. Partial reconfiguration enables dynamic replacement of parts of the softcore or even entire softcore processors at runtime. A practical evaluation of the functionality of the developed models as well an extensive array of experiments concerning the time requirements for the partial reconfiguration of softcore processors are presented. The practical results of this thesis clearly demonstrate the efficiency of developing solutions with the designed and realized toolchain, as well as the relevance and applicability of partial reconfiguration in the addressed domain.

# Vorwort

Diese Arbeit entstand an der Technischen Universität Ilmenau vor dem Hintergrund des von 2002 bis 2013 von der Deutschen Forschungsgemeinschaft geförderten Sonderforschungsbereichs 622 (SFB 622) „Nanopositionier- und Nanomessmaschinen“sowie 2013 bis 2014 und 2017 bis 2020 von der Thrüringer Aufbaubank geförderten Projekte „Spezifikation und Lösungen für Module und Teilsysteme in einem Framework zum Aufbau eines kompakten Weißlichtinterferometers für den industriellen Einsatz (WLI SoC)“ und „Oberflächenmesssystem zur sensornahen 3D-Topographieerfassung unter Einsatz kombinierter optischer Messverfahren für den produktionsnahen Einsatz (EKOM)“, in deren Rahmen das Fachgebiet Rechnerarchitektur und Eingebettete Systeme mit den Teilprojekten „Entwurf und Untersuchung applikationsspezidischer Hardware-Komponenten und System-on-Chip-Strukturen“ (WLI SoC) sowie „Erforschung und Entwicklung der sensornahen Informationsverarbeitung für einen Multisensorkopf zur 3D-Topographiemessung“ (EKOM) betraut war. Neben der Erstellung von spezifischen Hardware- und Software-Lösungen, die den Aufgabenstellungen aus den Bereichen der echtzeitkritischen Bildverarbeitung, Regelungstechnik und Hochpräzisionsmesstechnik gerecht werden sollten, stand die Erarbeitung von modellbasierten Methoden zum systematischen Entwurf und der Anpassung von Softcore Prozessoren als eingebettete und rekonfigurierbare (Teil-) Systeme im Vordergrund.
Die Entwicklung solcher Systeme unter der Maßgabe der harten Echtzeitfähigkeit bedeutet einen schwierigen Kompromiss zwischen echtzeitfähigen Speziallösungen, Methodenintegration, Wiederverwendbarkeit und Automatisierbarkeit einzelner modellbasierter Prozesse zur Beherrschung der Entwurfskomplexität.
Sowohl in der letzten Projektperiode des SFB 622 (von 2009 bis 2013), als auch in den Projekten WLI SoC und EKOM, traten Hardware-Plattformen auf Basis rekonfigurierbarer Logikbausteine in den Vordergrund. Mit der zunehmenden Leistungsfähigkeit dieser Module und den damit verbundenen System-on-Chip Technologien wurde die Realisierung komplexer integrierter Plattformen, applikationsspezifischer Hardwaremodule und Multiprozessorsysteme möglich.
Diese Arbeit fasst die methodischen Beiträge zur modellbasierten Entwicklung eingebetteter Softcore Prozessor-Lösungen in den oben genannten Projekten zusammen.
Um die Konsistenz mit den größtenteils englischsprachigen Quellen und damit etablierten Fachbegriffen zu erhalten, sind die Abbildungen, die in dieser Arbeit verwendet werden, in Englisch beschriftet. Entsprechende Erläuterungen und (sofern vorhanden) deutsche Entsprechungen sind den Texten zu entnehmen. Um die Lesbarkeit von zusammengesetzten Fachbegriffen in dieser Arbeit zu gewährleisten, wird auf den übermäßigen Gebrauch von Bindestrichen verzichtet, z. B. System-on-Chip Technologien anstatt System-on-Chip-Technologien. Weiterhin ist darauf hinzuweisen, dass die Begriffe Softcore und Softcore Prozessor im Rahmen dieser Arbeit als Synonyme voneinander verstanden werden.

# Inhaltsverzeichnis

# 1 Einleitung

Die zunehmende Leistungssteigerung bei gleichzeitiger Miniaturisierung digitaler Schaltkreise und die damit verbundene steigende Integrationsdichte von mikroelektronischen Schaltkreisen erlaubt die Realisierung von immer komplexeren und leistungsfähigeren Systemen. Aufgrund dieser immer weiter steigenden Komplexität ist es notwendig, neue Methoden und Verfahren zu entwickeln, die sich an die aufwendigere Entwicklung anpassen und mit entsprechend steigender Komplexität skalieren können.
Bereits einige Jahre nach der Erfindung der ersten integrierten Schaltungen veröffentlichte Gordon Moore eine Regel, bekannt unter dem Mooreschen Gesetz (Moore's Law), die die Entwicklung der Komplexität zukünftiger integrierter Schaltkreise abschätzt. Diese Regel besagt, dass sich die Funktionalität eines Chips etwa alle 24 Monate verdoppelt. [Moo98] Auch seit Beginn des 21. Jahrhunderts hat diese Regel weiterhin Bestand und wird auf absehbare Zeit weiterhin Bestand haben, wie die „International Technology Roadmap for Semiconductors (ITRS)“ [Sem09] zeigt. Diese sammelt die Daten von führenden Halbleiterherstellern der gesamten Welt, wertet diese aus und prognostiziert eine weitere Erfüllung dieser Regel. Die Entwicklung von Mikroprozessoren seit 1975 ist in Abbildung 1.1 dargestellt. Die steigende Integrationsdichte moderner Halbleitertechnologien ermöglicht sowohl die Realisierung von mehr Funktionalität, als auch den Betrieb bei höheren Taktfrequenzen. Dadurch können immer komplexere Probleme mit mikroelektronischen und damit auch eingebetteten Systemen gelöst werden, die bis dahin größeren Verarbeitungseinheiten vorbehalten waren. Gerade bei eingebetteten Systemen gilt es daher, einen guten Kompromiss zwischen Eigenschaften wie beispielsweise Performanz, Ressourcenverbrauch aber auch Leistungsaufnahme zu erzielen. Diese Eigenschaften sind zumeist gegenläufig, so wirkt sich z. B. die Steigerung der Performanz durch reine Erhöhung der Taktfrequenz negativ auf die Leistungsaufnahme aus.
Die steigende Komplexität eines solchen Systems erhöht im selben Maß auch die Entwurfkomplexität und damit die Entwicklungszeit. Dieser Trend wurde bereits Ende der 1960er Jahre erkannt. Es wurde nach Möglichkeiten geforscht, um die Herstellungs- und Entwicklungskosten zu verringern, Schaltungen zu verallgemeinern und trotzdem den speziellen Anforderungen der Kunden gerecht zu werden. In diesem Zusammenhang wurden erstmals allgemeine integrierte Schaltungen entwickelt, die einmalig beim Kunden ihre spezielle Logik „eingebrannt“ bekamen. Diese Entwicklung führte im Jahr 1972 zu einem Patent [YI72] über einmalig programmierbare Nur-Lese-Speicher (Programmable Read-Only Memory, PROM). Einmalig programmierbare Logikschaltungen bieten zwar einige Vorteile, waren aber nur der erste Schritt dieser Entwicklungskette, die im Jahr 1985 mit der Patentierung der heutzutage als FPGA (Field Programmable Gate Array) bekannten Technologie endete [PP85]. FPGAs sind die konsequente Weiterentwicklung und bieten die Möglichkeit, die interne Logik beliebig oft zu rekonfigurieren.
Mit Hilfe dieser Technologie ist es möglich, die Entwicklungszyklen drastisch zu reduzieren, da die entwickelte Logik ohne Zeitverzögerung getestet und etwaige Fehler sofort behoben werden können, ohne auf die Herstellung von Prototypen warten zu müssen. Diese Verzögerungen treten beispielsweise bei applikationsspezifischen Schaltungen, den ASICs (Application-Specific

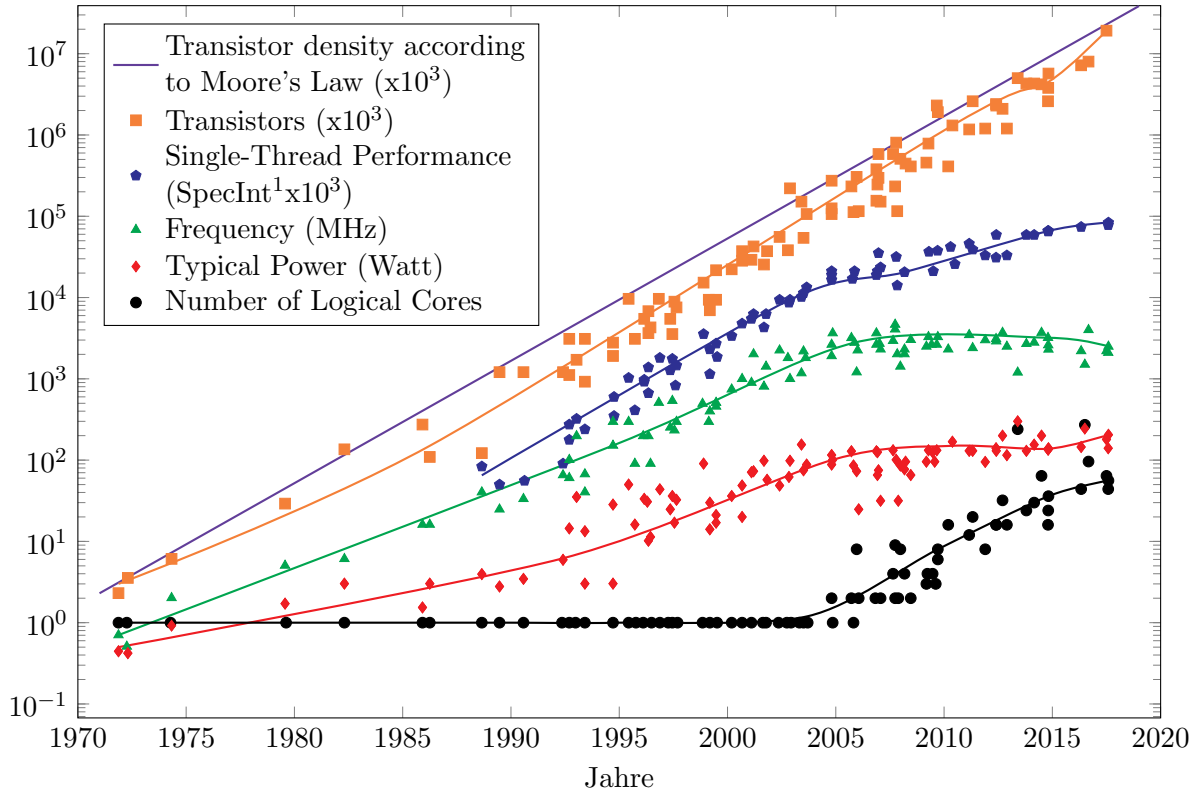



Abbildung 1.1: Entwicklung von Mikroprozessor Systemen[2]

Integrated Circuit), auf, was ein wesentlicher Nachteil ist.
FPGAs besitzen über 5,5 Millionen logische Zellen [Xil18c], was es den Entwicklern ermöglicht, extrem komplexe Schaltungen zu realisieren. Dabei werden auch die Leistungsanforderungen an diese Schaltungen immer größer. Beispielsweise verdoppeln sich die Geschwindigkeitsanforderungen, im Bereich drahtlose Kommunikation, etwa alle zwölf Monate (Hansen's Law) [BH03] und wachsen damit deutlich schneller als die Leistungsdichte von elektronischen Systemen nach dem Mooreschen Gesetz. Aus diesem Grund gewinnen Optimierungen, die vor allen Dingen die Verringerung des Energieverbrauchs und der Fläche bei steigenden Anforderungen (Rechenleistungen) zum Ziel haben, immer mehr an Bedeutung.

## 1.1 Echtzeitkritische Daten- und Bildverarbeitung

Unter dem Begriff der Echtzeitdatenverarbeitung wird ganz allgemein ein Bereich der Datenverarbeitung verstanden, bei dem die Ausführung eines Algorithmus innerhalb eines vorgegebenen Zeitfensters stattfinden muss. Zusätzlich existieren zeitliche und logische Abhängigkeiten zwischen der Ausführung dieser Rechenprozesse, deren Zeitfenster sich überlappen, d. h. die idealisiert nebenläufig ablaufen [Rze94].

[1] Benchmark Spezifikation nach [Sta17]
[2] Originaldaten bis 2010 aus [Sem09] zusammengestellt von M.Horowitz, F.Labonte, O.Shacham, K.Olukotun und C.Batten, 2010-2017 von K.Rupp

Aus dieser Definition lässt sich die Anforderung der Rechtzeitigkeit ableiten, die an Echtzeitsysteme zu stellen ist. Hierbei muss das Echtzeitsystem innerhalb eines deterministischen Zeitraums auf einen äußeren Stimulus reagieren und ein entsprechendes Ergebnis bereitstellen.
Eine Anwendungsdomäne mit speziellen Anforderungen stellt die Echtzeitbildverarbeitung dar. Hierbei wird ein digitales Bild, welches aus einer zweidimensionalen Anordnung von einzelnen Pixeln (Bildpunkten) besteht, algorithmisch verarbeitet. Um die gewünschten Informationen aus einem gegebenen Bild gewinnen zu können ist es notwendig, jedes Pixel des Bildes zu verarbeiten. Je nach vorgegebenem Algorithmus ist hier eine parallele Verarbeitung mehrerer Pixel zu einem Zeitpunkt möglich. Gerade bei echtzeitkritischen Problemen ist so eine Parallelisierung notwendig, um innerhalb der zeitlichen Vorgaben das gewünschte Ergebnis bereitstellen zu können. Dabei hat die Reihenfolge, in der die einzelnen Pixel verarbeitet werden, keinen Einfluss auf das Ergebnis der Bildverarbeitung. Eine mögliche Plattform zur Parallelisierung von Bildverarbeitungsalgorithmen stellen eingebettete Systeme, hier im speziellen die FPGAs, dar. Die eingebetteten Systeme im Allgemeinen werden im Abschnitt 2.1 und die FPGAs im Speziellen im Kapitel 2.4 detailliert vorgestellt.

## 1.2 Inhalt und Zielsetzung

Diese Dissertation beschäftigt sich mit einem wichtigen Teilgebiet der eingebetteten Systeme: Softcore Prozessoren auf FPGAs. Hierbei werden im speziellen Hochleistungsplattformen und deren Anforderungen adressiert. Zielsetzung dieser Arbeit ist die Entwicklung und Evaluation von Methodiken und Modellen zur Erstellung von Softcore Prozessoren und dazugehörigen Toolchains für SRAM-basierte FPGAs (vorgestellt in Abschnitt 2.4) als mögliche Verarbeitungseinheiten von Algorithmen. Als ein Beispiel für die Aufgabendomänen sei die echtzeitkritische Bildverarbeitung genannt. Die zentrale Softcoreeinheit wird hierbei als parametrisierbares Objekt angenommen, wie in Abbildung 1.2 schematisch dargestellt.

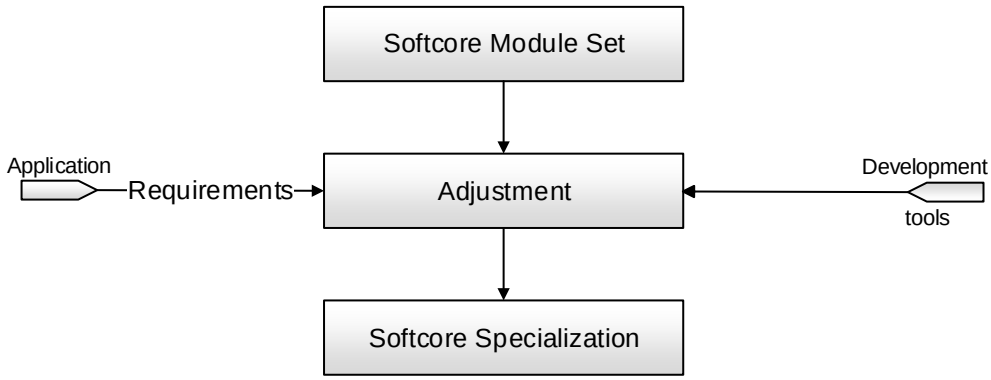



Abbildung 1.2: Softcore Spezialisierung

Das zweite Kapitel befasst sich mit allgemeinen Methoden zum Entwurf von eingebetteten Systemen mit FPGAs. Herausgearbeitet werden im speziellen der Stand der Technik und verschiedene Ansätze zu möglichen Entwurfsprozessen, im Hinblick auf den Einsatz eines Softcore Prozessors. Hierbei wird genauer auf den Systementwurf als Teil des komponentenorientierten

Entwurfs eingegangen. Es werden die notwendigen Grundlagen von FPGAs vermittelt, die speziell für das Verständnis der im späteren Verlauf angewandten Methoden benötigt werden. Weiterhin werden kurz System-on-Reprogrammable-Chip Technologien als Teil der System-on-Chip Technologien vorgestellt und wesentliche Merkmale genannt. Im Rahmen der eingebetteten Systeme mit FPGAs wird auf allgemeine und grundlegende Eigenschaften von notwendigen Übersetzungswerkzeugen eingegangen. Hierbei werden sowohl von Herstellern bereitgestellte Übersetzungswerkzeuge, aber auch eigene Entwicklungen eine Rolle spielen.
Die in dieser Arbeit adressierten Probleme der echtzeitkritischen Daten- und Bildverarbeitung, gefolgt von einer allgemeinen Bewertung des gesamten zweiten Kapitels, schließen dieses ab.
Das dritte Kapitel beschäftigt sich mit dem Entwurf von Softcore Prozessoren. Hierbei werden sowohl der Gesamtentwurf, aber auch der Entwurf einzelner Teile eines Softcores adressiert. Da Softcore Prozessoren eine Kombination aus Hard- und Software darstellen, wird ein Hardware-Software Co-Design Ansatz erläutert.
Das vierte Kapitel stellt die im Rahmen dieser Arbeit entwickelten und angepassten Methodiken und Modelle zum zeit- und kosteneffizienten Entwurf und Anpassung von Softcore Prozessoren im Rahmen spezifizierter Projekte der adressierten Aufgabenklasse dar. In diesem Zusammenhang wird zunächst ein spezielles Modell zur Softcoreentwicklung vorgestellt, gefolgt von einem angepassten V-Modell speziell zur Entwicklung mit System-on-Reprogrammable Chips. Aus den dargestellten Herausforderungen und Problemen wird eine allgemeine Toolchain zur Softcore und Softcore-Software Entwicklung abgeleitet, die in den folgenden Abschnitten detailliert vorgestellt wird. Der Fokus der gesamten Toolchain liegt auch hier auf einer effizienten Entwicklung mit Softcore Prozessoren. Entsprechende Konzepte zur Erstellung und Anpassung dieser beenden das vierte Kapitel.
Das fünfte Kapitel erbringt den Nachweis der korrekten Funktionalität der vorgestellten Toolchain, indem von jedem Verarbeitungsschritt dieser Kette eine praktische Realisierung vorgestellt, analysiert und auf verschiedene Leistungsmerkmale hin getestet wird. Ein spezielles Augenmerk bei der praktischen Realisierung ist die Anpassbarkeit und damit Wiederverwendbarkeit der realisierten Lösungen. In diesem Zusammenhang wird im Abschnitt der praktischen Realisierung des Softcore Prozessors speziell auf die Möglichkeit der Rekonfiguration des Prozessors zur Laufzeit der Lösung eingegangen.
Die Dissertation schließt mit einer Zusammenfassung sowie einem Ausblick für zukünftige weitere Arbeiten ab.

# 2 Aspekte des Entwurfs für Lösungen mit FPGA als Zielplattform

## 2.1 Eingebettete Systeme

Eingebettete Systeme sind Rechnersysteme, die als eine Komponente in eine Umgebung eingebettet sind und deren Aufgabe die (Teil-)Steuerung und/oder Überwachung der Umgebung ist. Dabei sind das Design, die Komplexität und der Zweck speziell auf diese Umgebung ausgelegt. Das eingebettete System unterliegt damit Bedingungen und Anforderungen, die auf eine spezielle Anwendung oder Klasse von Anwendungen beschränkt ist. [Mül14, BC12]

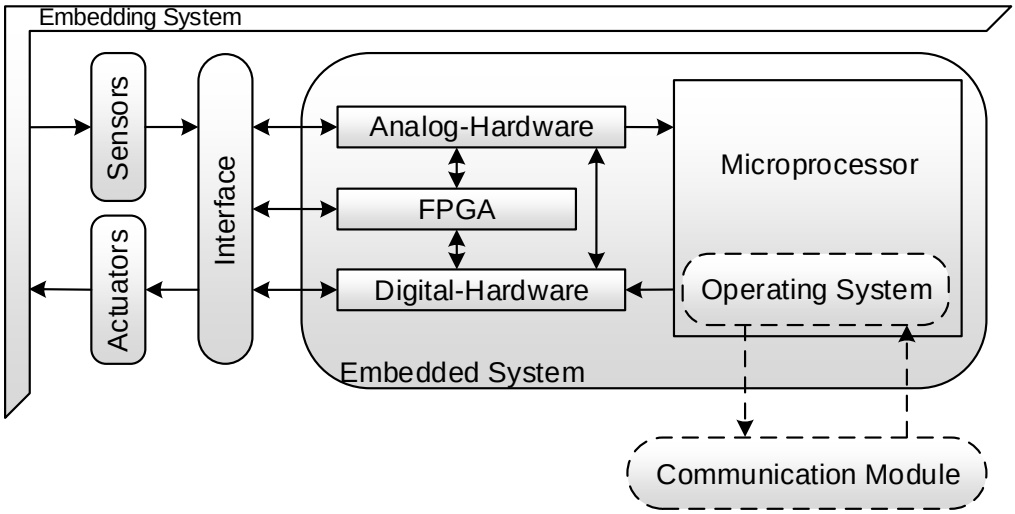



Abbildung 2.1: Eingebettetes System

Wie in Abbildung 2.1 dargestellt ist, findet eine Interaktion mit dem umgebenden System (*Embedding System*) über ein Sensor-Aktor Interface (*Sensors, Actuators, Interface*) statt. Hierbei erhält das eingebettete System (*Embedded System*) Umgebungsinformationen mittels Sensoren (*Sensors*) und reagiert über Aktoren (*Actuators*). Zusätzlich kann ein eingebettetes System über verschiedene Kommunikationsmodule (*Communication Module*) verfügen. Mit diesen ist eine Kommunikation mit anderen eingebetteten Systemen oder mit Menschen möglich. Letzteres wird als Mensch-Maschine-Schnittstelle bezeichnet und dient zur manuellen Parametrisierung und Überwachung. Grundsätzlich sind diese Systeme allerdings auf automatisiert ablaufende Prozesse ausgelegt, die keine menschliche Interaktion benötigen. Für die Lösung sehr komplexer Probleme kann es notwendig sein, mehrere eingebettete Systeme zu verknüpfen, sodass diese miteinander

interagieren können. Dies kann sowohl direkt oder auch in hierarchischen Strukturen erfolgen und benötigt ebenfalls entsprechende Schnittstellen. Ein wichtiger Bestandteil der meisten modernen eingebetteten Systeme, neben analoger und digitaler Hardware (*Analog-Hardware*, *Digital-Hardware*), sind FPGAs (*FPGA*). FPGAs stellen als Anordnung von programmierbaren Logikbausteinen (vereinfacht betrachtet) einen wichtigen Schwerpunkt dieser Arbeit dar.
Im Gegensatz zu normalen PCs (Personal Computer) besitzen eingebettete Systeme eine Reihe von Unterschieden, die sich aus ihrer Funktionsweise ableiten. Als exemplarische Beispiele können hier Größe oder physischer Formfaktor von medizinischen Geräten genannt werden, welche von Patienten getragen werden müssen. [BC12]
Eine wesentliche Eigenschaft und damit auch Unterschied ist der Energieverbrauch solcher Systeme. Dieser entwickelte sich in den vergangenen Jahren zu einer der dominierenden Charakteristika und wird in „*Thermal Design Power*“ (TDP) angegeben. In den meisten Einsatzfällen haben eingebettete Systeme einen sehr kleinen TDP von Mikrowatts bis zu wenigen Watts. Als Ergebnis müssen solche Systeme mit sehr aggressiven Energiesparmechanismen ausgestattet werden. [BC12, MKW11]
Aufgrund des „*Von Neumann Flaschenhalses*“ [BH03] ist die Speicherbandbreite ebenfalls ein wichtiger Faktor. Aus diesem Grund ist es notwendig, neue Entwurfsmethodiken und Verarbeitungsarchitekturen zu entwickeln. Eine allgemeine Entwurfsmethode für eingebettete Systeme und speziell für FPGAs stellt das V-Modell (nach [DW15]) dar und wird in Abbildung 2.2 gegeben.
Eine wichtige Aufgabe bei der Entwicklung von und mit eingebetteten Systemen ist das Anpassen von gegenläufigen Anforderungen. Hierbei spielen Eigenschaften wie Formfaktor, Ressourcen- und/oder Energieverbrauch, aber auch die Leistungsmerkmale wie Rechengeschwindigkeit eine wichtige Rolle. Diese Eigenschaften sind gegenläufig, d. h. eine Verbesserung der einen Eigenschaft führt in den meisten Fällen zu einer Verschlechterung einer anderen Eigenschaft. Aus diesem Grund ist es wichtig bei der Entwicklung von und mit eingebetteten Systemen die notwendigen Kriterien zwar zu erfüllen, aber möglichst nicht zu übererfüllen. Das ist auch der Grund, warum bei den meisten Projekten neue Systeme entwickelt werden müssen, obwohl es Lösungen gibt, die das gegebene Problem abdecken. Diese Lösungen sind nicht exakt auf das Projekt hin angepasst, decken daher meist ein nicht notwendiges Leistungsspektrum ab und daraus folgen ungewollte Nachteile, die eine Neuentwicklung notwendig machen.

Wie in Abbildung 2.2 zu sehen ist, müssen zu Beginn jeder Entwicklung eine Menge von Anforderungen (*Requitements*) und einschränkenden Eigenschaften (*Constraints*) definiert werden. Einschränkende Eigenschaften können hier beispielsweise bereits vorgestellte Eigenschaften, wie Energieverbrauch, sein. Normalerweise werden sowohl diese, als auch die Anforderungen extern, z. B. von einem Auftraggeber, bereit gestellt. Weitere wichtige Anforderungen für eingebettete Systeme, speziell FPGAs, sind:

- Funktionalitätsumfang
- Geschwindigkeitsanforderungen
- Echtzeitfähigkeiten
- maximale Kosten (Entwicklung.- und Hardwarekosten)
- Formfaktoren

Das allgemeine V-Modell stellt dabei einen sich immer weiter entwickelnden Standard dar, der begonnen von dem V-Modell 92, über das V-Modell 97 [Ver00], bis zu dem in Abbildung

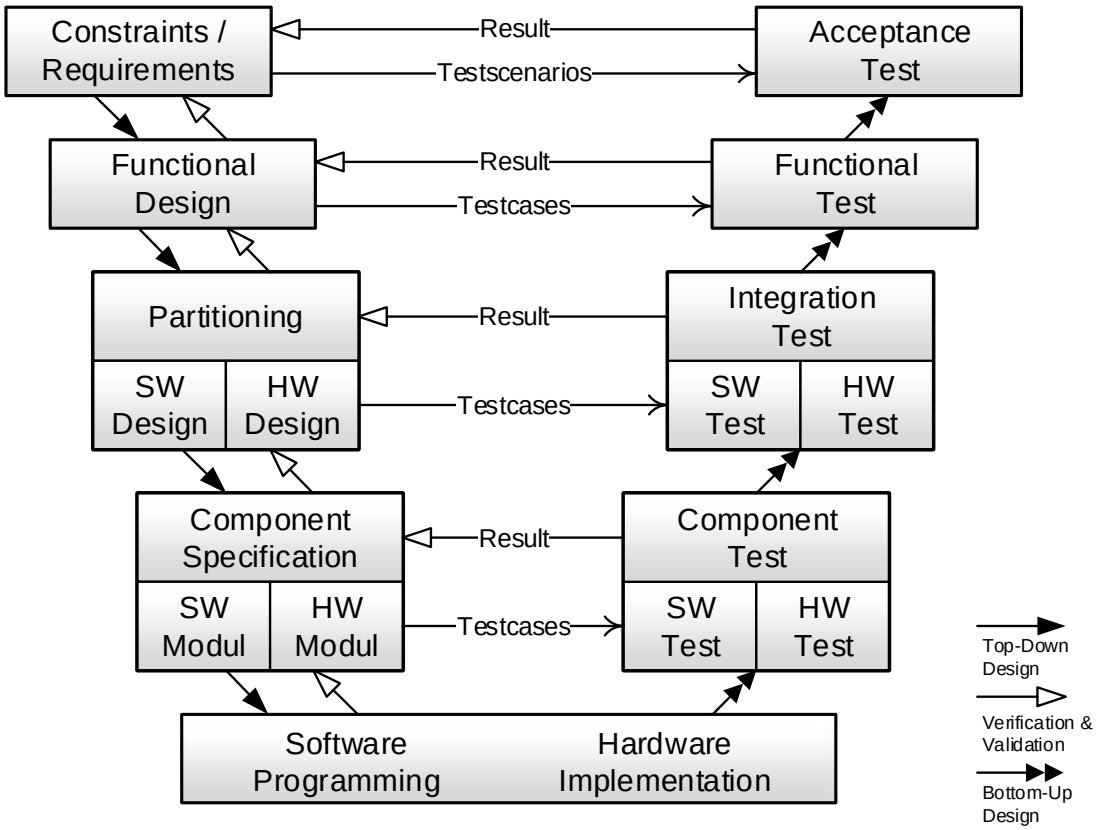



Abbildung 2.2: V-Modell für eingebettete Systeme

2.2 gezeigten angepassten V-Modell für eingebettete Systeme, immer wieder Änderungen speziell nach Einsatzschwerpunkt erhalten hat. Die in Abbildung 2.2 vorgestellte Methode spezifiziert vier Kernpunkte: das funktionelle Design (*Functional Design*), die Verfeinerung und Aufteilung (*Partitioning*) nach Hardware- und Software-Teilen (*HW Design*, *SW Design*), die Spezifikation der einzelnen Teilkomponenten (*Component Specification*) sowohl bei Hardware, als auch Software (*HW Modul* , *SW Modul*) und die physische Realisierung (*Implementation*) der Hardware, bzw. die Programmierung (*Programming*) der zugehörigen Software. Jeder dieser Kernpunkte wird gefolgt von einer auf dem selben Level befindlichen Testphase auf Einhaltung der Rahmenbedingungen und korrekte Funktionalität. In jedem Schritt ist eine Anpassung und/oder Verfeinerung möglich, die es gegebenenfalls notwendig macht, vorhergehende Schritte zu wiederholen. Am Ende der Entwicklung steht, je nach verwendeter Hardware, eine entsprechende (optionale) Herstellung. [Ash07]
Dieser Fall würde dann eintreten, wenn ein hochintegriertes Ein-Chip-System (System-on-Chip (SoC)) als applikationsspezifischer Schaltkreis (Applicationspecific Integrated Circtuit (ASIC)) entwickelt werden würde. Weiterhin gibt es Multi-Chip-Anordnungen (System-in-Package (SiP)), problemangepasste Mikrocontroller und Digitale Signalprozessoren (DSP) (System-on-Board (SoB)), sowie universelle embedded-PCs mit Echtzeitbetriebssystemen. [Mül14, VG06]
Die Aufteilung in Hardware- und Softwarekomponenten ist besonders bei Entwicklungen mit FPGAs ein wichtiger Schritt und sollte deshalb, wie in der Abbildung 2.2 gezeigt, modellgestützt erfolgen. Eine Möglichkeit ist eine toolgestützte Umsetzung und Erweiterung des V-Modells, wie

beispielsweise das V-Modell XT. [FKSH09]
Wie bereits erwähnt, hat das allgemeine V-Modell immer wieder Anpassungen, je nach Einsatzschwerpunkt erfahren. Die innerhalb des Modells angestrebte Aufteilung (*Partitioning*) in Hardware und Software ist zu allgemein um sinnvoll mit der immer weiter steigenden Komplexität der eingebetteten Systeme skalieren zu können. Eine verfeinerte Aufteilung ist notwendig. Weiterhin ist es bei der Verwendung dieses Modells nicht möglich, auf die Gegebenheiten spezieller Teile der eingebetteten Systeme sowie deren Konzeption und Realisierung einzugehen.

## 2.2 Systementwurfsmodelle

Aufgrund von einer nicht abschätzbaren Anzahl von Einflussfaktoren kann kein allgemein gültiges Vorgehensmodell bei der Entwicklung neuer Systeme gegeben werden. Vielmehr gibt es eine große Anzahl von Methoden und Techniken, die sich nach [BVK08] in vier (Haupt-) Kategorien klassifizieren lassen:

### Sequentielle Vorgehensmodelle

Modelle dieser Familie ordnen den Entwicklungsaktivitäten Phasen zu. Diese werden sequentiell durchgeführt und es wird vorausgesetzt, das bei einem Phasenübergang die vorangegangene Phase abgeschlossen ist. Als Beispiele können Phasenmodell [Ben83], Wasserfall- oder Schleifenmodell [Roy87] genannt werden.
Das V-Modell ordnet sich ebenfalls in die Kategorie der Sequentiellen Vorgehensmodelle ein.

### Prototypische Vorgehensmodelle

Als Prototypische Vorgehensmodelle werden sequentielle Modelle bezeichnet, die zu definierten Zeitpunkten der Entwicklung kontrollierte Rückschritte und Wiederholungen vorangegangener Phasen festlegen. Diese Zeitpunkte werden dabei definiert durch die Entwicklung von Prototypen, d. h. eingeschränkt funktionsfähige oder vereinfachte Versionen des geplanten Systems. Ein Beispiel hierfür ist das Spiralmodell [Boe88].

### Wiederholende Vorgehensmodelle

Im Gegensatz zu den Sequentiellen Vorgehensmodellen werden in dieser Familie die entsprechenden Phasen wiederholt durchlaufen. Ausgehend von einer Teilmenge aller Anforderungen werden bei diesen Modellen Teilergebnisse in sequentieller Reihenfolge erstellt. Bei jedem erneuten Durchlauf wird die Teilmenge der Anforderungen erweitert und es wird ein erneuter Durchlauf gestartet. Als Beispiele sind hier alle inkrementellen, evolutionären, rekursiven und iterativen Vorgehensmodelle zu nennen, beispielsweise das iterative Vorgehensmodell nach [BT75].

#### Wiederverwendungsorientierte Vorgehensmodelle

Bei dieser Kategorie handelt es sich um Modelle, die auf Wiederverwendung von Ergebnissen aus vorangegangenen Projekten setzen. Gleichzeitig wird bei allen Neuentwicklungen darauf geachtet, die Wiederverwendbarkeit für zukünftige Projekte zu maximieren. Als Beispiel können Hier alle domänenorientierten Vorgehensmodelle [EF04] angeführt werden.

Für die Entwicklung von eingebetteten Systemen werden angepasste Vorgehensmodelle mit Top-Down und Bottom-Up Mechaniken verwendet. Bei einem Top-Down Entwurf findet zunächst eine Spezifikation der Anforderungen statt. Darauf aufbauend werden Funktionsbeschreibungen entwickelt. Im Anschluss daran wird die eigentliche Struktur aus miteinander kommunizierenden Teilschaltungen entwickelt. Dieses Entwurfsverfahren wird in Kapitel 2.3.1 detaillierter vorgestellt. Der Bottom-Up Entwurf agiert in entgegengesetzter Richtung. hier werden aus Bauteilen mit bekannter und getesteter Funktionalität größere Funktionseinheiten zusammengesetzt. [Kem11]

## 2.3 Komponentenorientierter Entwurf

Der allgemeine komponentenorientierte Entwurf ist der erste Ansatz der Massenproduzierbarkeit von Software, beschrieben von McIlroy [McI68]. In diesem Zusammenhang wird unter Massenproduzierbarkeit sowohl sehr große Projekte, als auch eine Vielzahl von ähnlichen Projekten verstanden. Bei diesem Entwicklungsansatz werden objektorientierte Mechanismen für Aufbau und Wiederverwendung mit Eigenschaften wie Gleichzeitigkeit oder Verteiltheit integriert. Ein wichtiger Ansatzpunkt hierbei ist das dynamische Updaten von eingebetteten Systemen, das eine Reihe von Vorteilen bietet. Programme können Laufzeit-optimiert, Fehler behoben oder Funktionalitäten erweitert werden. [VB02]
Nach [IEE90] ist eine Komponente ein Teilstück, aus dem ein (Gesamt-)System besteht, wobei diese Komponente sowohl aus Hardware, als auch aus Software bestehen kann. Weiterhin kann diese intern in andere Komponenten unterteilt sein.

### 2.3.1 Der Top-Down Systementwurf

Bei einem Top-Down Entwurf werden zunächst die Anforderungen (*Requirements*) und Randbedingungen (*Constraints*) spezifiziert und Entwurfsziele definiert. Mit Hilfe dieser wird ein Gesamtsystem (*System Specification*) beschrieben. Im Entwurf von eingebetteten Systemen mit FPGAs sollte an dieser Stelle eine Notationsbeschreibung (engl. *denotational description*), also ein Set von Gleichungen und Ungleichungen in einer geeigneten Algebra angefertigt werden. Mittels dieser kann die korrekte Funktionalität bereits weitgehend vor der eigentlichen Implementierung und Programmierung überprüft werden. Das Endziel des Designers ist ein Gesamtsystem dessen Verhalten exakt diese Notationsbeschreibungen widerspiegelt und dabei die definierte Menge von Anforderungen und Randbedingungen einhält. Dank der frühen Überprüfbarkeit lassen sich potenzielle Designiterationen und damit Entwicklungszeit und Kosten sparen. [SVM01]
Darauf aufbauend werden schrittweise einzelne Funktionsbeschreibungen (*Functional Specification*) gebildet. Dies kann zunächst informell geschehen und wird im weiteren Prozess immer weiter formalisiert. Am Ende dieser Phase sollte ein formal beschriebenes Funktionsmodell

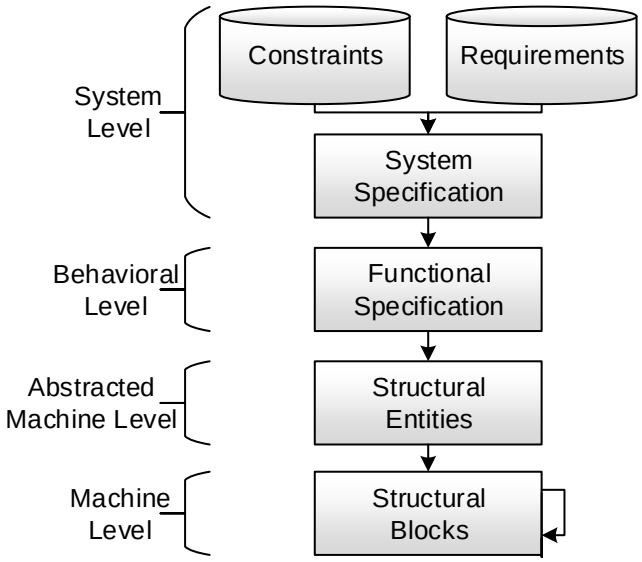



Abbildung 2.3: Allgemeiner Top-Down Systementwurf

existieren, welches simulierbar ist. Alle Spezifikationen sowohl im Systemlevel (*System Level*), als auch auf dem Verhaltenslevel (*Behavioral Level*) sind plattformunabängig. Mit diesen Funktionsbeschreibungen ist eine Evaluierung der Beziehungen der einzelnen Designvariablen möglich. Beispiele für diese Funktionsbeschreibungen in der FPGA-Entwicklung, die eine verfeinerte Stufe der Notationsbeschreibungen darstellen, sind Datenfluss Modelle, Event-getriebene Modelle oder Finite State Machine (FSM) Spezifikationen.
Im weiteren Verlauf des Top-Down Systementwurfes werden auf der Hardwareseite miteinander kommunizierende Teilschaltungen und auf der Software Seite miteinander agierende Teilprogramme (*Structural Entities*) nachgebildet. Diese befinden sich auf dem Register-Transfer Level, bzw. werden mit einer Hochsprache realisiert und sind in Abbildung 2.3 als abstrahiertes Maschinenlevel (*Abstracted Machine Level*) dargestellt. Mit weiterer Verfeinerungen gehen diese in das Maschinenlevel (*Machine Level*) über. Unter Maschinenlevel werden in diesem Zusammenhang alle Verfeinerungen verstanden, die mit der Kombination aus hardwareseitigem Logik Level und softwareseitigem Maschinencode enden. Die kleinsten Einheiten im Top-Down Entwurf bilden die Funktionsblöcke (*Structural Blocks*). Diese können rekursiv weitere Funktionsblöcke enthalten, die je nach Verfeinerungsstufe bis auf Gatter oder Transistorebene, bzw. Binärcode herab gehen können. [LL13, Kem11, VG06]
Mit Verfahren dieser Art ist es möglich, eingebettete Systeme nach speziellen Anforderungsszenarien zu entwickeln. Aufgrund der Verfeinerungsstruktur muss sich auf eine Realisierungsplattform erst spät im Entwicklungsprozess festgelegt werden. Die Wahl einer speziellen FPGA-Familie ist hierbei beispielsweise erst nach ersten Simulationen notwendig, wenn die Rahmenbedingungen wie z. B. Ressourcenverbrauch und benötigte Chipfläche bereits sehr genau abgeschätzt werden kann.
Die Entwicklung von eingebetteten Systemen rein nach diesem Systementwurf weist aber ebenfalls eine Reihe von Nachteilen auf. Verglichen mit der Entwicklung von traditioneller Computersoftware gibt es so gut wie keine Wiederverwendungsmöglichkeiten von bereits

entwickelten, teilweise plattformabhängigen, Funktionsblöcken. Dies führt zu längeren und damit teureren Entwicklungszyklen. Optimierungen hinsichtlich wichtiger Eigenschaften wie z. B. Größe oder Stromverbrauch können erst sehr spät im Entwicklungsprozess vorgenommen werden und können gegebenenfalls zu einer Verletzung des Anforderungsprofils führen, da zu Beginn der Entwicklung diese Faktoren nicht absehbar sind.

### 2.3.2 Der Bottom-Up Systementwurf der rekonfigurierbaren Ebene

Bei einem Botton-Up Entwurfskonzept allgemein wird der Entwurf in entgegengesetzter Richtung durchlaufen. Das bedeutet Bauteile, also einzelne Funktionsblöcke, zu größeren Funktionseinheiten zusammengesetzt werden. Dies bietet verschiedene Vorteile, da es mit diesem Ansatz möglich ist, einzelne Blöcke hinsichtlich ihrer

- Größe
- Geschwindigkeit
- Stromverbrauch

hin zu optimieren. Ein weiterer Vorteil ist die sofortige Testbarkeit der korrekten Funktionalität. Bei diesem Entwurfsverfahren ist es allerdings schwierig, vorgegebene Zielfunktionen, also komplette Gesamtsysteme nachzubilden. [LL13, Kem11] Dieser Entwicklungsprozess basiert auf Expertenwissen über die speziellen Anforderungen der jeweiligen Anwendungsdomäne. [LTT11] Bei dem Bottom-Up Systementwurf des V-Modells (Abbildung 2.2, rechte Seite) enthält jede Phase die Erzeugung von ausführbaren (Teil-)Systemen aus dem Code und der Integration des ausführbaren Systems in die Umgebung des FPGA, bzw. des eingebetteten Systems. [LL13] Diese Art des Systementwurfs ist notwendig, wenn gewisse spezifizierte Funktionen durch spezielle Implementierungskomponenten ersetzt werden müssen, da beispielsweise ansonsten eine mangelnde Ausdrucksmächtigkeit entsprechenden Entwicklungsebene vorliegt. Hierdurch können zusätzliche, eventuell nicht abbildbare, Funktionalitäten als externe Komponenten auf Implementierungsebene eingefügt werden. Der Vorteil der separierten Entwicklung von Komponenten zum modellbasierten Systementwurf entsteht durch die Komponentenabstraktion und die Integration in Bibliotheken, die als Black-Box-Blöcke für die Erstellung von projektspezifischen Modellen im Zuge der Entwicklung in den Gesamtentwurf eingefügt werden können. [Mül14]
Bei einem Bottom-Up Systementwurf der rekonfigurierbaren Ebene von FPGAs werden einzelne Designmodule entwickelt. Hierbei ist zu beachten, dass für jede Partition dieser Module keine Optimierungen über die physischen Grenzen der jeweiligen Blöcke vorgenommen werden können. Jedes Modul muss unabhängig von umgebender Logik entwickelt und synthetisiert werden. Allerdings nutzen alle (partiellen) Module das gleiche Top-Level sowie gleiche Platzierungs- und Routing-Ergebnisse. Hierbei werden entsprechend unabhängige, separierte Netzlisten erzeugt. Die Top-Level Logik muss dabei mit entsprechenden Black-Box Modulen für die jeweiligen partiellen Teile vorgesehen werden. [Xil17c]

## 2.4 Field Programmable Gate Arrays

Das Konzept der vorliegenden Arbeit basiert auf den Möglichkeiten heutiger FPGAs. FPGAs gehören zu der Gruppe der programmierbaren Logikschaltkreise. Programmierbar bedeutet in diesem Zusammenhang, dass komplexe Hardwarefunktionalitäten über in einer Hardwarebeschreibungssprache (Hardware Description Language (HDL)) geschriebene Sprachelemente realisiert werden. Diese Konfigurierbarkeit und vor allem die Rekonfigurierbarkeit, bleibt auch nach der Integration in das eingebettete Zielsystem erhalten, was den wesentlichen Vorteil von FPGAs im Vergleich zu anderen Technologien, wie z. B. ASICs, ausmacht. [Gro11]
FPGAs werden mit unterschiedlichen Hardwarekonfigurationen, Konfigurierungsmöglichkeiten und Einsatzgebieten bis hin zu strahlungsresistenten FPGAs für Luft- und Raumfahrt von diversen Herstellern angeboten, beispielsweise Xilinx, Intel (ehemals Altera), Atmel oder Microsemi.
Die Designentscheidung einen FPGA zu verwenden bietet diverse Vorteile. So ist es aufgrund der Rekonfigurierbarkeit möglich, wesentlich kürzere Designzyklen zu realisieren. Fehler in einer implementierten Hardware können sehr schnell und ohne zusätzliche Kosten behoben werden. Der Designprozess eines ASIC benötigt die Herstellung von Prototypen, die in vielen Fällen mehrere Monate in Anspruch nimmt und mit Herstellungskosten verbunden ist. Ein FPGA kann im Vergleich innerhalb von Minuten neu Konfiguriert und erneut getestet werden, ohne das Herstellungskosten anfallen. Entsprechend werden in Projekten, in denen FPGAs die verwendete Hardwareplattform darstellt, deutlich weniger Kosten generiert.

### 2.4.1 Funktionsweise von Field Programmable Gate Arrays

Die Möglichkeit anwendungsspezifische Schaltungen auf dem Chip auszutauschen wird durch eine generische, feingranulare Grundstruktur im Silizium erreicht, in der eine große Anzahl von Logikblöcken (*Logic Block*) in einer prorammierbaren Verbindungsmatrix (*Programmable Interconnect*) angeordnet sind. Zusätzlich sind diese von programmierbaren Eingabe/Ausgabe-Blöcken (*Input/Output Block*) an der Chip-Peripherie umgeben. Die Logikblöcke können dabei konfigurierbare Logikzellen enthalten, aber auch Speicherblöcke oder dedizierte Berechnungsressourcen wie Multiplizierer. Eine Logikzelle enthält dabei eine Anordnung aus allen Grundelementen. [Gro11, KTR08]

Ein beispielhafter Aufbau ist in Abbildung 2.4 dargestellt und ist [Nat12] nachempfunden. Hierbei können die konfigurierbaren Logikblöcke (Configurable Logic Block (CLB)) aus mehreren Look-Up Tables (LUT), Speicherzellen und Verknüpfungselementen aufgebaut sein. Der Aufbau der CLBs unterscheidet sich nach Hersteller und FPGA-Familie. [Mei10]

### 2.4.2 Partiell Rekonfigurierbare FPGA-Plattformen

Eine Untermenge der FPGAs bilden die FPGA-Plattformen mit der Fähigkeit zur partiellen Rekonfiguration (partial Reconfiguration (PR)). Unter partieller Rekonfiguration wird die teilweise Rekonfiguration von FPGA Chipfläche über partielle Bitfiles zur Laufzeit des FPGA verstanden. Die wichtigste Eigenschaft hierbei ist, dass die nicht rekonfigurierte Chipfläche während des gesamten Rekonfigurationsvorgangs weiter operabel bleibt. Die wichtigsten

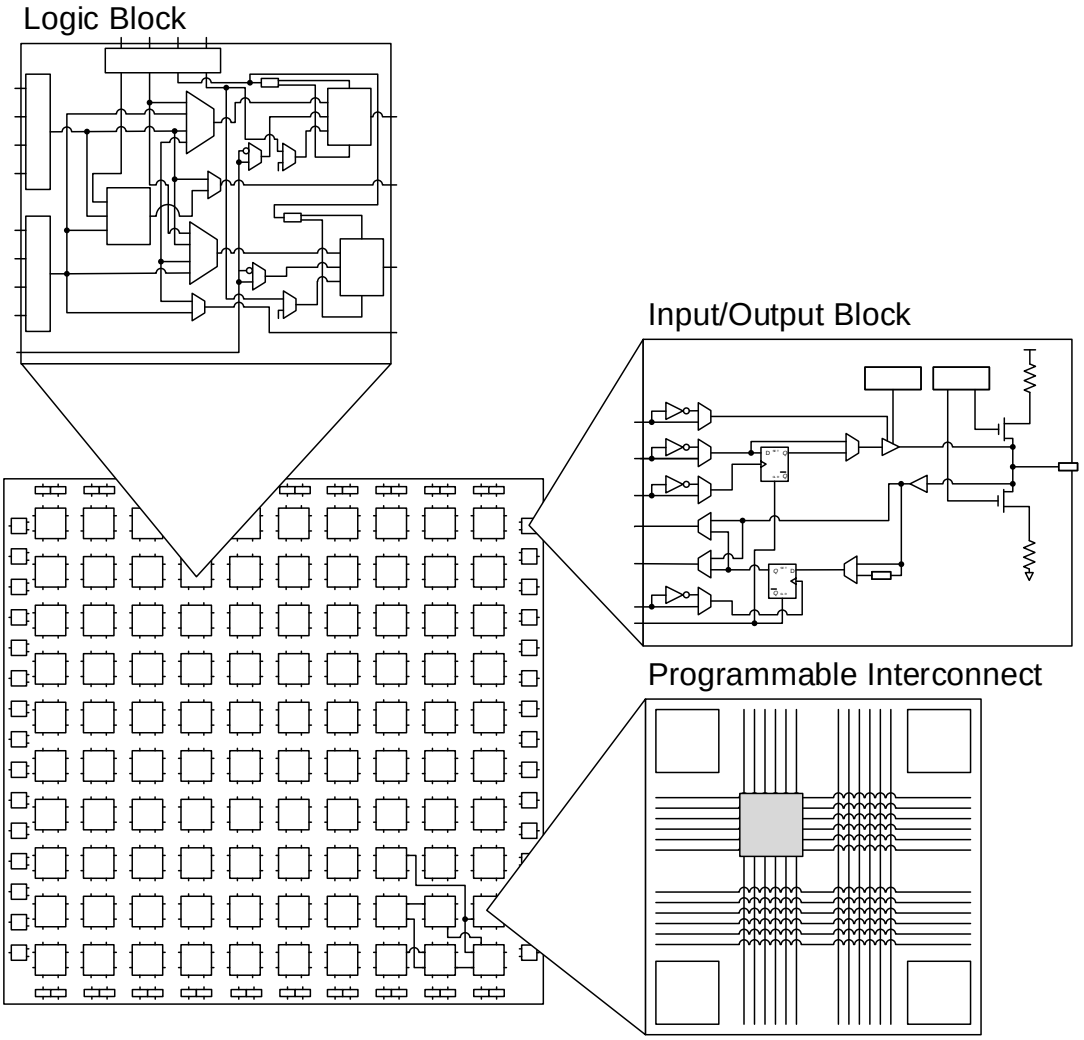



Abbildung 2.4: FPGA Aufbau (nach [Nat12])

Ressourcen, wenn es um partielle Rekonfigurierbarkeit geht, sind die bereits beschriebenen CLBs und BRAMs sowie zusätzlich DSP Blöcke und Input/Output Blöcke (IOBs).
Verschiedene FPGA-Hersteller bieten Chips mit der Fähigkeit zur partiellen Rekonfiguration an. Dabei hat jeder Hersteller leicht voneinander abweichende Bezeichnungen für die Möglichkeit, einen Teil eines FPGAs zu rekonfigurieren, während der restliche Chip operabel bleibt. Exemplarisch werden von der Firma Xilinx der Begriff „dynamisch partielle Rekonfiguration“ [Xil17c], oder von der Firma Intel (ehemals Altera) der Begriff „hierarchisch partielle Rekonfiguration“ [Int18] verwendet. Im weiteren Verlauf der Arbeit werden diese Begriffe unter dem Begriff partielle Rekonfiguration (partial Reconfiguration (PR)) vereinheitlicht.
Zum Zeitpunkt dieser Arbeit sind lediglich die zwei genannten Hersteller Intel und Xilinx bekannt, die mit aktuellen FPGA-Familien das Feature der partiellen Rekonfiguration unterstützen. Weitere Hersteller wie National Semiconductor, Lattice Semiconductor und Atmel haben die partielle Rekonfiguration in früheren FPGAs implementiert, aktuelle Familien

dieser Firmen unterstützen dieses Feature allerdings nicht mehr. Aus diesem Grund entfallen diese Hersteller für den in der Arbeit präsentierten Teil der Methoden, die eine partielle Rekonfiguration benötigt.
Bei der partiellen Rekonfiguration werden also FPGA Ressourcen innerhalb einer zeitlichen Domäne gemultiplext, was je nach Einsatzszenario enorme Vorteile in Bezug auf Ressourcen und Energieverbrauch bringen kann. Aufgrund der Heterogenität der vorhandenen Literatur erfolgt im folgenden eine für die Zwecke dieser Arbeit geeignete Zusammenfassung. Um eine partielle Rekonfiguration von Teilen des FPGAs, zu ermöglichen, ist es notwendig das Layout (und damit den Chip) in vordefinierte Regionen aufzuteilen. Die im folgenden vorgestellten Methoden können nur auf einer Designebene angewandt werden, auf der ein FPGA-Chip, bzw. zumindest eine FPGA-Familie innerhalb eines Projektes bereits festgelegt wurden, da diese Methode von der physischen Realisierung des speziellen Chips abhängig ist.

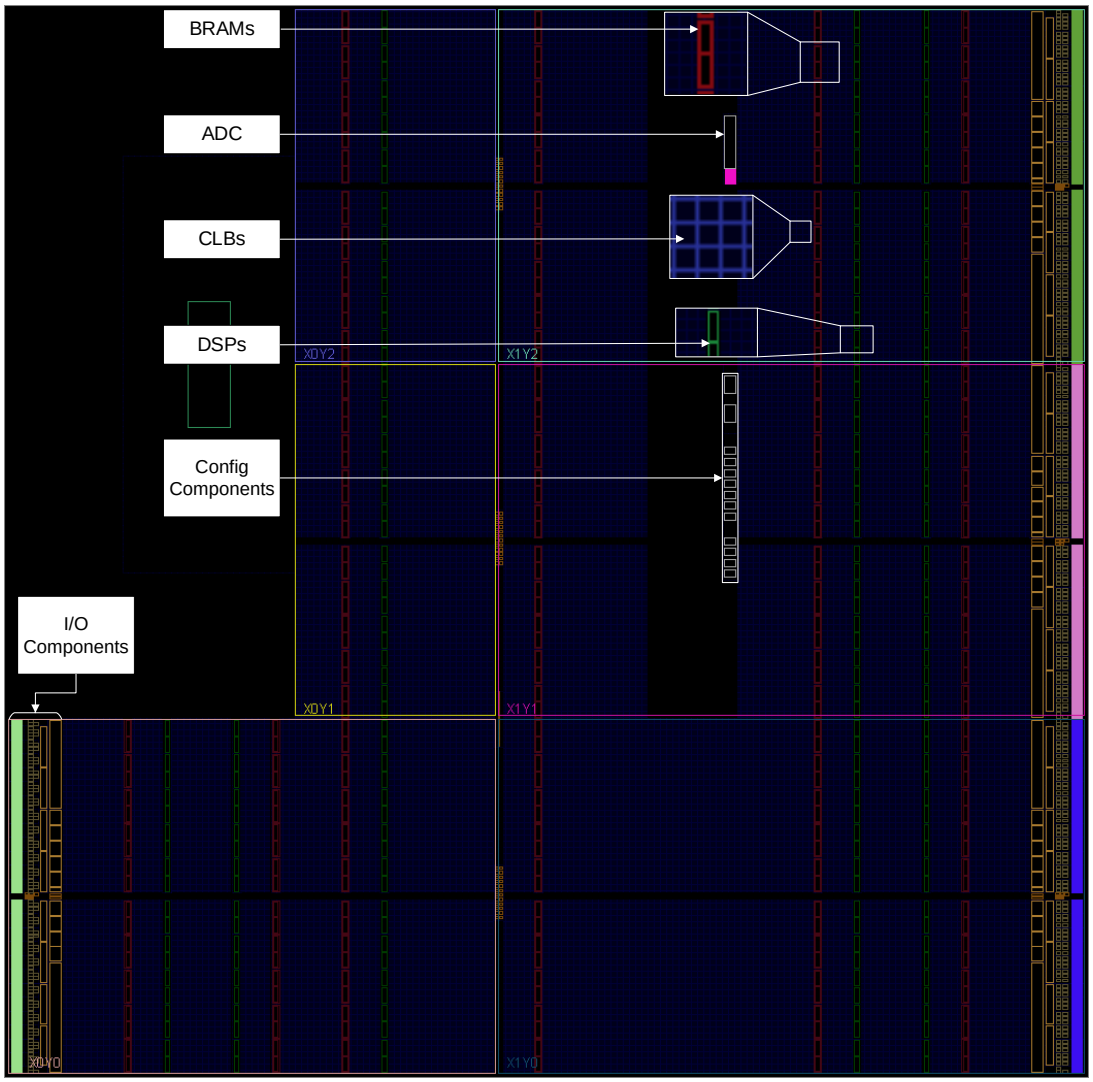



Abbildung 2.5: Exemplarischer FPGA-Aufbau

In Abbildung 2.5 sind die für die partielle Rekonfiguration zu beachtenden Bauteile auf der hier betrachteten FPGA Familie hervorgehoben. Zu sehen sind (nach [Xil13]):

- Programmierbare Logikblöcke (*Configurable Logic Blocks* (*CLBs*)), auch Slices genannt, die aus verschiedenen Look-Up Tabellen, Multiplexern oder Shift-Registern bestehen können und als blaue Rasterblöcke angeordnet sind.
- Verschiedene Speicherblöcke (*BRAMs*), in rot dargestellt, die aus FiFo Block RAM Speicher, allgemeinen RAM Blöcken oder auch Read-Only Memory (ROM) bestehen können und in mehreren senkrechten Linien angeordnet sind.
- Arithmetische Funktionsblöcke (*DSPs*), in grün dargestellt, die ebenfalls in mehreren senkrechten Linien angeordnet sind.
- Verschiedene Konfigurationsschnittstellen (*Config Components*), wie beispielsweise DNA-Port, Configuration-Data-Access-Port oder Configuration-Frame-Error-Correction-Port.
- Eingabe-/Ausgabekomponenten (*I/O Components*), beispielsweise Input/Output-FiFos, Input-Buffer oder I/O-Bänke.

Die Rekonfiguration erfolgt, genau wie die initiale Konfiguration, über Bitströme (*Bitstreams*), die über eine definierte Schnittstelle in den Konfigurationslayer geschrieben werden. Im Fall der partiellen Rekonfiguration wird von partiellen Bitströmen gesprochen und diese setzt immer eine zuvor erfolgte initiale Konfiguration voraus. [Cla11]
Ein SRAM basierter FPGA kann als zwei-Layer Modell betrachtet werden: funktionaler Layer (*Functional Layer*) und Konfigurationslayer (*Configuration Layer*). Ein Zugriff ist nur auf den Konfigurationslayer möglich und wirkt sich direkt auf den funktionalen Layer aus. Der Konfigurationslayer ist in eine Chip-abhängige Anzahl an Konfigurationsreihen (*Configuration Rows*) in vertikaler Richtung und in Konfigurationsframes (*Configuration Frame*) in horizontaler Richtung eingeteilt. Ein Konfigurationsframe entspricht der kleinsten rekonfigurierbaren Einheit und wurde in Abschnitt 2.4.2 vorgestellt.

Wie in Abbildung 2.6 zu sehen ist, sind Konfigurationsframes (*Configuration Frame*) für die Konfiguration von bestimmten Elementen im funktionalen Layer zuständig. Abhängig von dieser funktionalen Zuordnung werden bestimmte Konfigurationsframes zu funktional gleichen Konfigurationssektionen (*Configuration Column*) zusammengefasst. Obwohl jedes Konfigurationsframe aus mehreren Speicherzellen (*SRAM Memory Cell*) aufgebaut ist, ist eine Manipulation einzelner Speicherzellen nicht möglich, da diese nicht separiert adressiert werden können. [Cla11]
Für jede in Abbildung 2.5 zu sehende Ressource gibt es eigene Konfigurationssektionen auf dem Konfigurationslayer des FPGAs. Dabei ist die Anzahl der Konfigurationsframes in jeder Konfigurationssektion abhängig davon, für welchen funktionalen Block die jeweilige Sektion zuständig ist. Weiterhin variiert die Anzahl an Konfigurationsframes von funktionsgleichen Konfigurationssektionen zwischen verschiedenen FPGAs, aufgrund der Variation der Komplexität des jeweiligen FPGAs.

Bei der partiellen Rekonfiguration werden vordefinierte Areale eines FPGAs abgeschaltet und neu konfiguriert, während der restliche Chip operabel bleibt.
Dazu ist es notwendig während des Designprozesses diese Areale zu definieren. Hierbei müssen verschiedene Aspekte in Betracht gezogen werden, die im Folgenden diskutiert werden sollen.
Zu beachtende Eigenschaften der Areale sind: die Lage des Areals auf dem Chip, die Lage

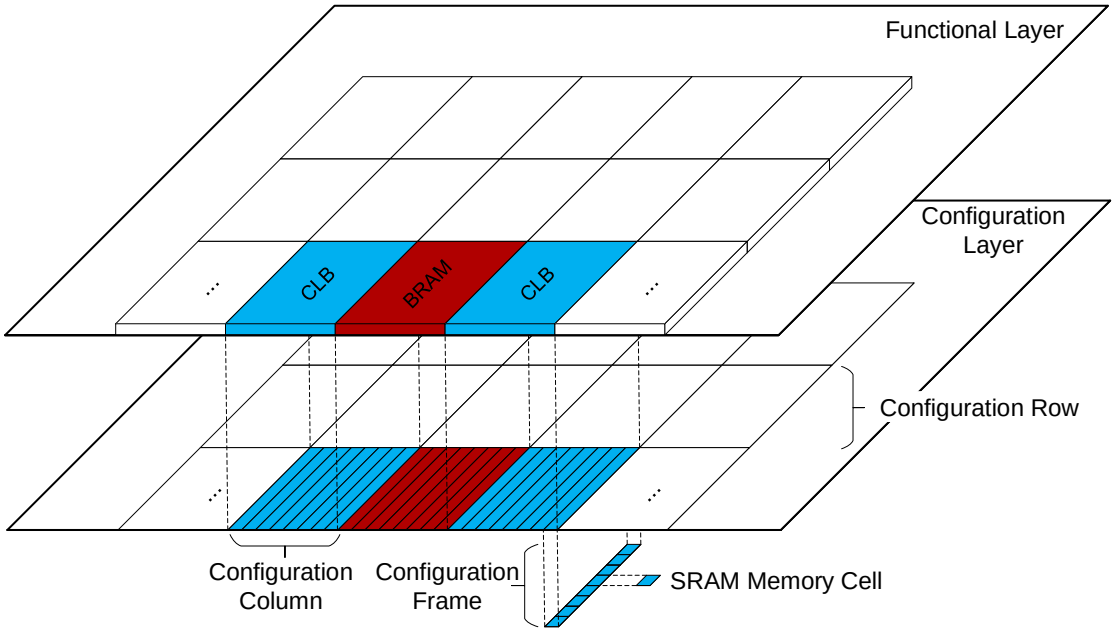



Abbildung 2.6: Konfigurationslayer und funktionaler Layer (angelehnt an [Cla11])

relativ zu einem anderen Areal, Ressourcen die innerhalb dieser Fläche und zu realisierende Logiken im Areal. Weiterhin ist es wichtig das sämtliche auf dieser Fläche realisierte Logik die selben I/O Schnittstellen zu der umliegenden Logik verwendet. Das bedeutet, zum Zeitpunkt der Festlegung der Lage und der Fläche des Areals alle IP-Komponenten (*Intellectual Property* (*IP*)) die dort im späteren Verlauf eingebunden werden sollen bereits realisiert sein müssen. Die IP-Komponenten jedes voneinander getrennten Konfigurationszustands werden im folgenden als partielle Rekonfigurationskomponenten (*Partial Reconfiguration Componentent* (*PRC*)) bezeichnet.

Die Anzahl der verwendeten Konfigurationssektionen von CLBs, BRAMs, DSP, usw. hängt von dem maximalen Ressourcenbedarf aller in einem Areal einzusetzenden PRCs ab. Dabei müssen alle Konfigurationssektionen unabhängig voneinander betrachtet werden. Wie in Abbildung 2.7 über die Größe der PRCs „A1“ bis „A3“ und „B1“ sowie „B2“, sowie in folgender Tabelle exemplarisch illustriert.

| PRC | CLB | DSP | BRAM |
|---|---|---|---|
| A1 | 100 | 3 | 5 |
| A2 | 80 | 2 | 0 |
| A3 | 50 | 0 | 0 |
| B1 | 150 | 2 | 2 |
| B2 | 50 | 5 | 2 |

Tabelle 2.1: Beispiel zur Rekonfiguration mit mehreren PRCs (zu Abbildung 2.7)

Zu sehen ist, dass für den Block A die PRC „A1“ die Größe des festzulegenden Areals definiert. Es werden mindestens 100 Einheiten CLBs, drei DSPs sowie fünf Einheiten BRAMs innerhalb des Areals benötigt.

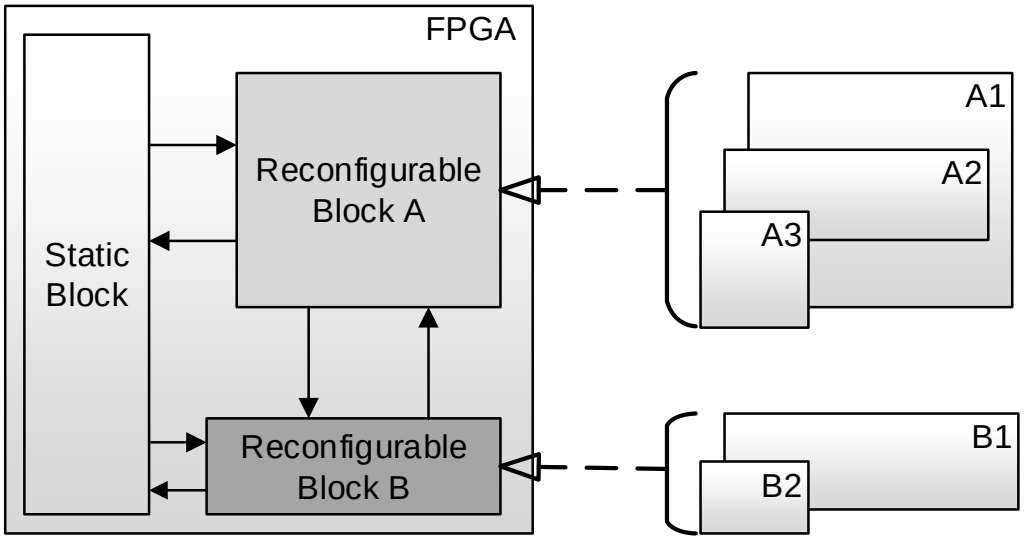



Abbildung 2.7: FPGA mit statischem und partiell rekonfigurierbaren Teilen

Für den Block B werden insgesamt mindestens 150 Einheiten CLBs, fünf DSPs sowie zwei Einheiten BRAMs benötigt, da zwar Block „B1“ mehr CLBs benötigt als Block „B2“, aber dieser mehr DSPs verwendet. Das resultierende Areal muss also sowohl mindestens 150 Einheiten CLBs zur Verfügung stellen, als auch mindestens fünf DSPs, sowie zwei BRAMs.
Bei zu rekonfigurierenden Regionen ist es also sinnvoll Blöcke gegeneinander auszutauschen, die einen möglichst ähnlichen Ressourcenbedarf haben, da die Grenzen der zu rekonfigurierenden Regionen nicht dynamisch veränderlich sind.
Es ist weiterhin möglich, dass jeder rekonfigurierbare Teil mit dem statischen Teil kommuniziert, aber auch mit einem anderen rekonfigurierbaren Teil. Eine Kommunikation ist dabei nur während des Betriebs einer Konfiguration möglich und nicht während der Rekonfiguration.

Die zu realisierenden Logikmodule definieren also die minimalen Ressourcen, die innerhalb eines Areals verfügbar sein müssen. Auf Basis dieser benötigten Ressourcen können mögliche Platzierungen des Areals auf dem Chip ermittelt werden, wie in Abbildung 2.8 illustriert wird. So sollten beispielsweise Module die der Kommunikation mit Chip-externen Modulen dienen in die Nähe entsprechender I/O-Ressourcen gelegt werden. Es ist allerdings auch wichtig, die Pfadlängen zu anderen Ressourcen wie beispielsweise internen Taktgeneratoren bei der Auswahl mögliche Platzierungen zu berücksichtigen, genauso wie die lokale Nähe zu anderen Modulen mit denen eine direkte Kommunikation notwendig ist. Es entstehen also verschiedene mögliche Platzierungsoptionen (*Placement Options*), die anhand von gezeiteten Modellen (*Timing Model*s) evaluiert werden müssen.
Das Optimierungsziel sollte hier eine Minimierung der maximalen Pfadlänge und damit Signallaufzeit sein. Eine minimierte kritische Signallaufzeit führt zu einer Erhöhung der maximalen Taktrate und damit zu einer kürzeren Abarbeitungszeit bei gleichem Ressourcenbedarf.

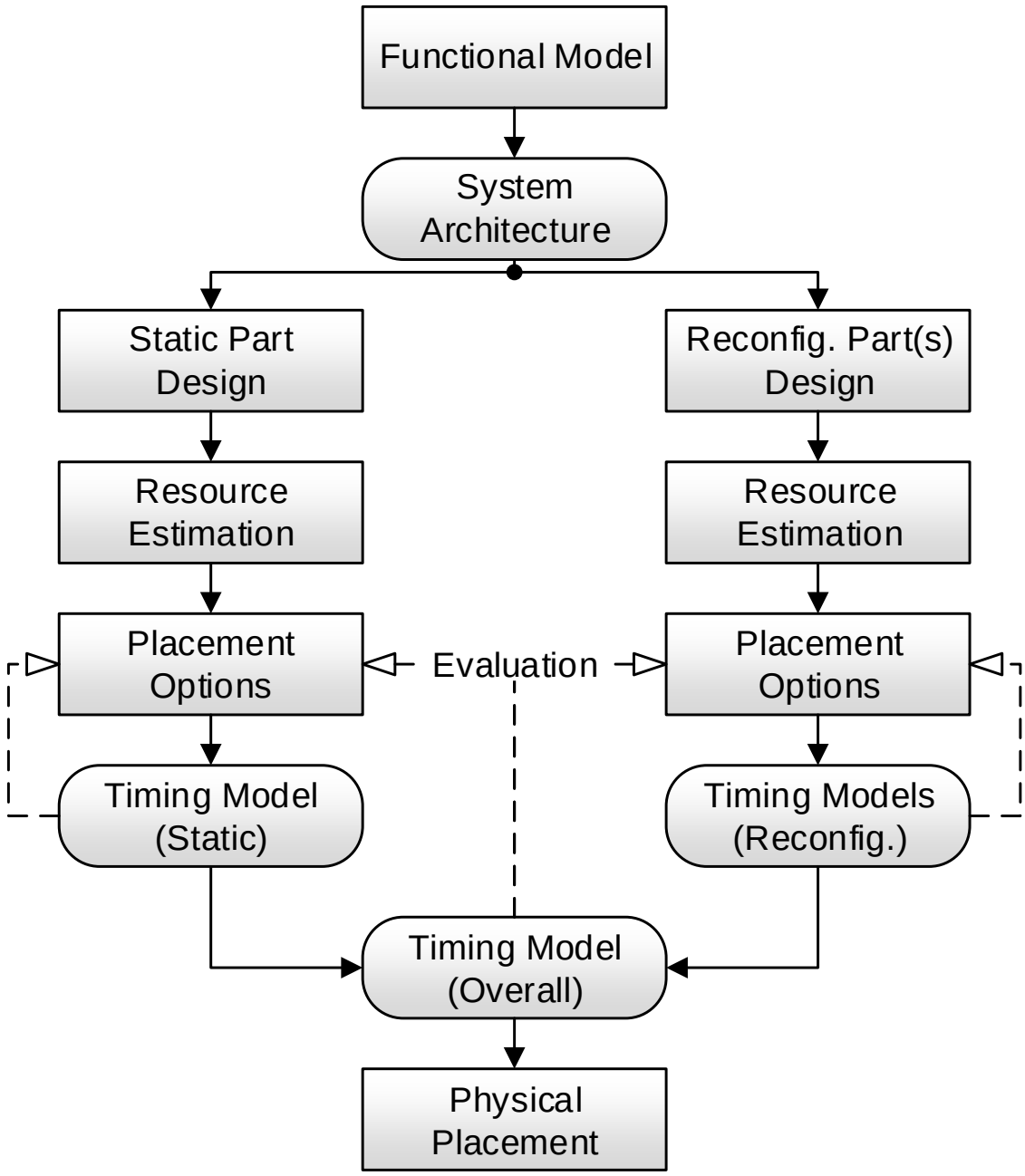



Abbildung 2.8: Schema zur Logikplatzierung mit rekonfigurierbaren Teilen

#### Konfigurationsframes

Alle programmierbaren Einheiten innerhalb eines FPGA werden über flüchtige Speicherzellen (*volatile memory cells*) konfiguriert. Diese müssen bei jedem Start neu konfiguriert werden und werden im allgemeinen als Konfigurationsspeicher (*configuration memory*) bezeichnet. Diese definieren die Gleichungen der LUTs, das Signalrouting und die Input/Output Blockansteuerung und weitere Funktionalitäten. Dieser Konfigurationsspeicher ist in Kategorien eingeteilt, sogenannte Frames. Diese Frames stellen die kleinsten adressierbaren Segmente dar. Alle

Operationen müssen daher auf ganzen vielfachen dieser Konfigurationsframes (*Configuration Frames*) ablaufen. [Xil17c]
In allem FPGAs der 7-Series, also auch der Zynq-7000 Familie, werden die für die Rekonfiguration verwendeten Rekonfigurationsframes aus diesen Konfigurationsframes aufgebaut. Dabei setzt sich ein Rekonfigurationsframe aus den in Tabelle 2.2 dargestellten Anzahl an Blöcken zusammen.

| Baustein | Höhe | Breite |
|---|---|---|
| CLB | 50 | 1 |
| DSP46 | 10 | 1 |
| BRAM | 10 | 1 |

Tabelle 2.2: Bausteinanzahl für Rekonfigurationsframe (nach [Xil17c])

Das bedeutet, dass ein Rekonfigurationsframe immer mindestens aus beispielsweise einer 50 Zeilen langen Spalte an CLBs und je einer zehn Zeilen langen Spalte an DSP46 und BRAMs aufgebaut sein muss, sofern diese Ressourcen verwendet werden sollen. Dieses Konzept kann auf beliebige reokfigurierbare FPGA-Familien anhand der Datenblätter und Rekonfigurationsframe-Größe angepasst werden.

### Rekonfigurationsgranularität

Eine wichtige Einteilung innerhalb der rekonfigurierbaren Bausteine ist die Granularität mit der diese die Rekonfiguration vornehmen können. Hierbei wird zwischen feiner Granularität (*fine-grained*) und grober Granularität (*coarse-grained*) unterschieden. Dabei wird unter feiner Granularität die Rekonfigurierbarkeit basierend auf Verarbeitungselementen, die auf Bit-Level arbeiten, verstanden. Unter Rekonfiguration mit grober Granularität wird die Ersetzung von komplexen Funktionsblöcken, wie beispielsweise ganzen „Arithmetischen-Logischen-Einheiten“ (*Arithmetic Logic Unit* (*ALU*)) verstanden. Besonders bei dieser Granularität zeigen sich große Performanz und Effizienz Vorteile, gemessen an Energieverbrauch oder benötigter Chipfläche. Werden innerhalb eines FPGA zur Optimierung von Energieverbrauch oder Erweiterung der Chipfläche verschiedene hard-IPs (*Intellectual Property* (*IP*)) verwendet, so wird von einer gemischen Granularität (*mixed-grained*) gesprochen. [Koc14]

### Vor- und Nachteile von partieller Rekonfiguration

Die Idee, FPGA Ressourcen innerhalb einer zeitlichen Domäne zu multiplexen, birgt großes Potenzial beim Einsparen von Ressourcen und Energie. Allerdings gibt es dabei auch einige Dinge zu beachten, die im Folgenden diskutiert werden sollen.
Eine Rekonfiguration benötigt Zeit, in der die zu rekonfigurierende Logik nicht arbeiten kann. Wie viel Zeit für eine Rekonfiguration benötigt wird, hängt von folgenden Faktoren ab:

- der verwendeten FPGA-Familie und damit der maximalen Konfigurationsgeschwindigkeit dieser (siehe dazu Grafik 2.9);
- Größe der zu rekonfigurierenden Region (und die dadurch bestimmte Größe der Bitstreams);

- der jeweilige Speicherlösung der Bitstreams und ob diese genügend Geschwindigkeit ermöglicht um die maximal mögliche Konfigurationsgeschwindigkeit auszunutzen.

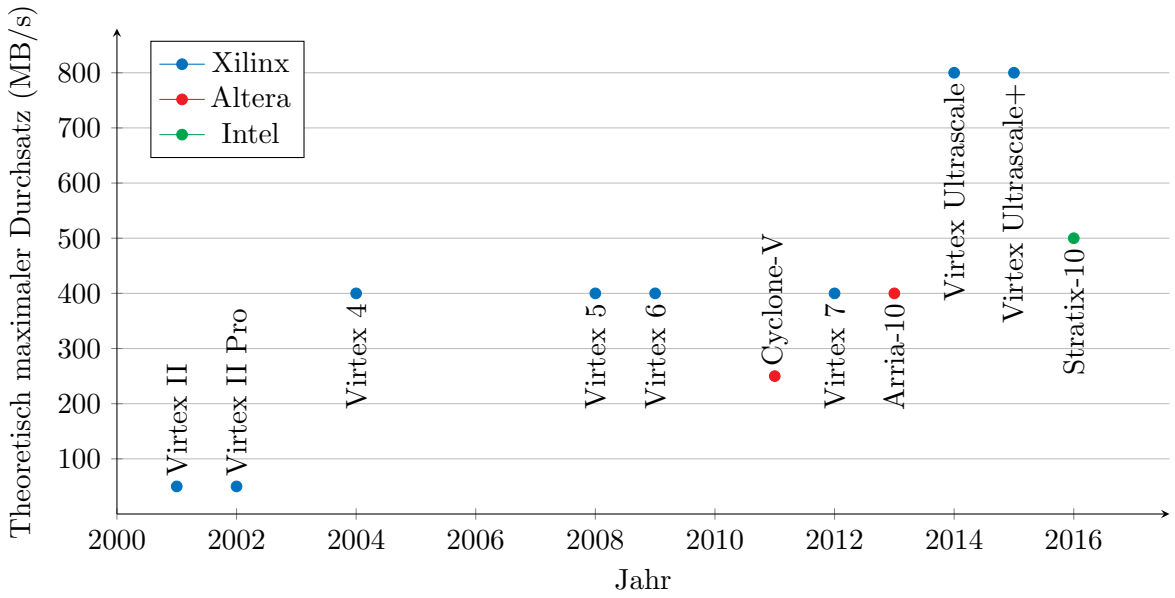



Abbildung 2.9: Maximaler Rekonfigurationsdurchsatz

In Grafik 2.9 zu sehen ist die Entwicklung des maximalen Rekonfigurationsdurchsatzes über verschiedene FPGA-Familien der Firmen Xilinx und Intel (ehemals Altera). Da diese beiden Unternehmen die einzigen FPGA-Hersteller sind, die partielle Rekonfiguration auch in modernen FPGA-Familien anbieten, werden im Folgenden nur noch diese beiden Firmen betrachtet, da die Möglichkeit einen FPGA partiell zu Rekonfigurieren von zentraler Bedeutung in dieser Arbeit ist.
Wie zu sehen ist, ist die Steigerung des maximalen Rekonfigurationsdurchsatzes sehr gering für einen Entwicklungszeitraum von 14 Jahren. Besonders wenn in Betracht gezogen wird, dass sich die verfügbaren Ressourcen auf diesen FPGAs, und damit auch die Größen der benötigten Bitstreams vervielfacht haben. So benötigt der Virtex II zwischen 0,04 - 3,35 MB [Xil14a] für einen Bitstream der den gesamten FPGA konfiguriert (je nach FPGA der speziellen Virtex II-Familie) und der Virtex Ultrascale+ bis zu 113,32 MB. [Xil18d]
Wird jetzt davon ausgegangen, dass von beiden FPGAs jeweils 10 % der Chipfläche rekonfiguriert werden sollen, ergibt das eine Rekonfigurationszeit des Virtex II von 0,08 ms ($0,04\ \text{MB} \cdot 0,1 \div 50\ \text{MB/s} = 0,08\ \text{ms}$), im Vergleich zu 14,17 ms ($113,32\ \text{MB} \cdot 0,1 \div 800\ \text{MB/s} = 14,17\ \text{ms}$) Rekonfigurationszeit des Ultrascale+. Das bedeutet, dass die Ressourcen der FPGAs deutlich schneller steigen als die Geschwindigkeiten, mit denen sowohl Konfigurationen, als auch Rekonfigurationen vorgenommen werden können.
Um eine partielle Rekonfiguration durchführen zu können, müssen zusätzliche Komponenten in das Design eingebaut werden, die zusätzliche Ressourcen des FPGAs belegen. Sollte sich ein Design selbst rekonfigurieren können, wird ein Rekonfigurationscontroller (mehr dazu in Abschnitt 4.4.4) zur Bereitstellung der partiellen Bitströme benötigt. Weiterhin müssen diese partiellen Bitströme gespeichert werden.
Dank des steigenden maximalen Rekonfigurationsdurchsatzes werden die Möglichkeiten partielle Bitströme zu speichern und mit ausreichender Geschwindigkeit wieder zu laden geringer. Die Anforderungen an den Speicher werden höher und damit werden bisherige Speicherlösungen, wie

beispielsweise reine Flash-Speicher, ungeeignet.
Eine detaillierte Diskussion der Speicherlösung im Rahmen der umgesetzten praktischen Lösung der vorliegenden Arbeit erfolgt in Abschnitt 5.1.3.

### 2.4.3 System-on-Chip Technologien

Das stetig steigende Leistungspotenzial der Halbleitertechnologien macht es notwendig, neue Modelle zur Produktivitätssteigerung zu entwickeln. Ein wichtiger Ansatzpunkt hierbei ist die Verringerung der Entwurfskomplexität und des Reimplementierungsaufwands beim Chip-Entwurf. Dies kann vor allem durch großflächige Wiederverwendung von bereits existierenden Komponenten und Infrastrukturen erfolgen. Ein Verbund von derartigen Komponenten, die über standardisierte Schnittstellen kommunizieren und auf einem Chip realisiert sind, wird als hochintegriertes Ein-Chip-System (*System-on-Chip* (*SoC*)) bezeichnet. [SWM+06]
Dabei umfasst der Begriff SoC generell alle Hardwareentwürfe und schließt damit auch ASICs und direkt in Silizium realisierte Schaltkreise mit ein. Geht man nun davon aus, dass innerhalb des Systems immer mindestens ein FPGA verwendet wird, so wird von einem System-on-Reprogrammable-Chip (*SoRC*) gesprochen. [Mül14, Xil10]
Durch den stark komponentenorientierten Ansatz wird die Entwicklung solcher Systeme in die zwei bereits vorgestellten Verfahren unterteilt: die Bottom-Up Entwicklung der IP-Komponenten und die Top-Down Integration eines anwendungsspezifischen SoC innerhalb einer Realisierungsarchitektur.

#### IP-Cores als SoC-Komponenten

IP-Cores sind fertig entwickelte und getestete Komponenten, die als (Teil-) Komponenten in den Gesamtentwurf eingebunden werden können. Mögliche IP-Core Komponenten können Prozessoren, Speicherbaugruppen, Schnittstellenkontroller oder nutzerspezifische Funktionsblöcke sein. Es werden drei Gruppen von IP-Cores unterschieden: [CH11, SWM+06]

- Soft-IPs - auf HDL-Ebene anpassbar
- Firm-IPs - nur parametrisierbar
- Hard-IPs - nur als Black-Box integrierbar

Um IP-Cores möglichst vielseitig wiederverwenden zu können ist eine standardisierte Schnittstelle, also ein möglichst generisches Interface (siehe Abbildung 2.10), notwendig. Dabei kann verallgemeinert gesagt werden, das jede Schnittstelle einen Datenpfad (*Data IN & OUT*) und einen Steuerpfad (*Control IN & OUT*) besitzen muss. Hierfür ist meist eine Anpassungslogik (*Interface Adaption Logic*) notwendig. Zusätzlich besteht bei getakteten IP-Cores sowohl die Möglichkeit, den externen Takt zu verwenden (*Bus CLK*) oder einen eigenen von dem Bus abweichenden Takt (*Local CLK*). Als IP-Core Wrapper wird die Anpassung von Signalen aus Daten und Steuerpfad aus einem generischen Interface in das vom IP-Core verwendete Interface inklusive der dafür benötigten Adaptionslogik bezeichnet. Die Definition von universellen IP-Core Schnittstellen ist Gegenstand aktueller Forschung, da gerade entsprechende Adaptionslogiken einen zusätzlichen Overhead im Sinne des Ressourcenverbrauchs bedeuten und etwaige Taktraten einschränken können.

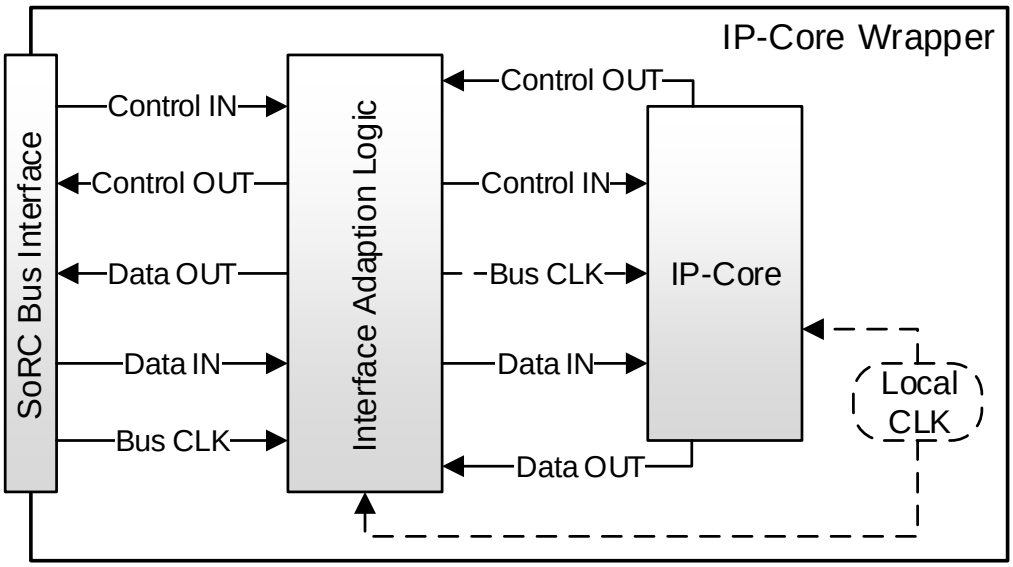



Abbildung 2.10: IP-Core mit Wrapper an SoRC Bus Interface

#### Plattform-FPGAs

Die Kombination aus SoC und FPGAs wird als Plattform-FPGA bezeichnet. Eine wesentliche Eigenschaft sind die auf dem Chip fest integrierten Teilsysteme aus Prozessorkernen (*Processor Core*), Speicherkontrollern (*Memory Controller*), Speicher (*On Chip Memory*, *Off Chip DDR-RAM*) sowie standardisierten Schnittstellenkontrollern (*MUX IO*, *AXI Bus*) aus. Eine exemplarische Darstellung ist in Abbildung 2.11 gegeben.

### 2.4.4 Ressourcenverbrauch und Energieeffizienz

Zwei der wichtigsten Faktoren bei der Entwicklung von SoRCs sind Ressourcenverbrauch und Energieeffizienz. Diese stehen mit anderen Faktoren, wie beispielsweise einer geringen time-to-market (*TTM*) Zeit, in Konflikt. Dabei spielt das geplante Einsatzgebiet eine entscheidende Rolle. Viele SoRCs und damit auch FPGAs werden in mobilen Geräten eingesetzt, die nicht an eine permanente Stromversorgung angeschlossen sind, also mit einem begrenzten Vorrat an Energie auskommen müssen. Zusätzlich ist der Formfaktor ein wichtiges Kriterium. Soll das System z. B. als medizinisches Implantat eingesetzt werden, so spielt die Größe und damit der Ressourcenverbrauch eine entscheidende Rolle. Diese zwei Kriterien sollen im Folgenden genauer betrachtet werden:

#### Ressourcenverbrauch

Einer der großen Vorteile bei der Entwicklung mit FPGAs, verglichen mit anderen Technologien wie ASICs, ist die sehr kurze Entwicklungszeit und damit die geringen Entwicklungskosten

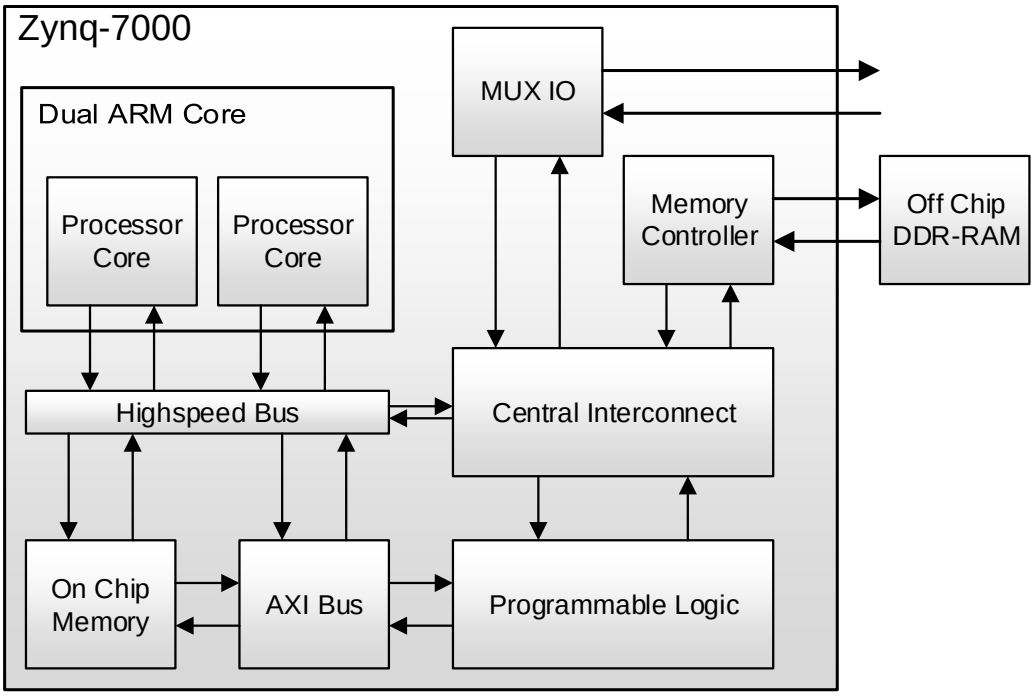



Abbildung 2.11: Plattform-FPGA Xilinx Zynq-7000 Familie

aufgrund der Rekonfigurierbarkeit. Es findet keine zeitaufwendige Herstellung von zu entwickelnder Hardware statt, was die Designzyklen stark verkürzt. Dieser Vorteil wirkt sich allerdings negativ auf die Performanz aus und erhöht die benötigte Chipfläche. In [KR07] wurde der Ressourcenverbrauch als benötigte Chipfläche bei gleicher Realisierung sowohl auf einem 90nm SRAM FPGA, als auch auf einem 90nm ASIC verglichen. Als Vergleichsgrundlage dienten hier verschiedene Algorithmen, umgesetzt auf dem FPGA in reiner Logik, Logik und vorkonfigurierten Arithmetikschaltungen (vom Hersteller als DSP bezeichnet), Logik und Speicher, sowie Logik, Speicher und „DSPs“. Das geometrisch gemittelte Ergebnis kann Tabelle 2.3 entnommen werden. Die dort angegebenen Werte entsprechen den Faktoren, die die FPGA Realisierung mehr Fläche als die funktionsgleiche ASIC Logik benötigt.

| Logik | Logik & DSP | Logik & Speicher | Logik, Speicher & DSP |
|---|---|---|---|
| 35 | 25 | 33 | 18 |

Tabelle 2.3: Ressourcenvergleich FPGA-ASIC im geometrischen Mittel (aus [KR07])

#### Energieeffizienz

Ein weiterer wichtiger Aspekt bei der Entwicklung von eingebetteten Systemen mit FPGAs ist der erhöhte Energiebedarf einer funktionell gleichen Realisierung, verglichen mit anderen Technologien wie ASICs. Hierbei muss zwischen statischem und dynamischem Energieverbrauch

unterschieden werden. Unter statischem Energieverbrauch wird dabei der Strombedarf verstanden, den eine Schaltung während Inaktivität benötigt. Meist wird das mit dem Stromdurchfluss in Transistoren im inaktiven Stadium assoziiert.
Unter dynamischem Energieverbrauch wird dabei der Verbrauch des Designs während einer Aktivität verstanden. Beeinflusst wird dieser durch die Menge an Daten, die das Design in einem gegebenen Zeitintervall berechnen muss, beispielsweise über das Schalten von Transistoren. [MKW11]
Es ist möglich, beide Faktoren während der Entwicklung auf den verschiedenen Entwicklungsebenen zu beeinflussen. Auf dem Architekturlevel können Designentscheidungen und Optimierungen der Energiekontrolle mittels der verwendeten Applikationssoftware beeinflusst werden. Nach der Systempartitionierung ist es möglich, Teile des Designs in unterschiedlichen Clock-Domänen zu betreiben. Eine niedrigere Frequenz sorgt aufgrund des dynamischen Energieverbrauchs auch für eine geringere Gesamtleistungsaufnahme. Allerdings verringert eine niedrigere Frequenz auch die Verarbeitungsgeschwindigkeit und damit den maximalen Durchsatz.
Auf dem Implementierungslevel sind, je nach verwendeter Software, Compiler und Übersetzer, weitere architekturspezifische Optimierungen über spezielle Implementierungstechniken umsetzbar. [MKW11]
Weitere Möglichkeiten die Energieeffizienz zu erhöhen sind die Verwendung von Dynamic Voltage Scaling (*DVS*) oder Dynamic Power Management (*DPM*). DVS nutzt dabei die Tatsache, dass der Energieverbrauch von CMOS Prozessoren sich quadratisch mit der Versorgungsspannung erhöht. Wird die Versorgungsspannung nun gesenkt, verringert den Energieverbrauch quadratisch, wohingegen sich die Laufzeit der jeweiligen Schaltung nur linear verringert. [Mar17]
DPM bezeichnet einen Ansatz, bei dem Prozessoren verschiedene Energiesparstadien verwenden. Wichtig für die Effektivität des DPM ist die Entscheidung, wann in einen Energiesparmodus geschaltet wird. Hierfür gibt es verschiedene Ansätze: einfache Timer, welche die Transistoren für eine bestimmte Zeit in den Standby-Modus schalten, andere Ansätze modellieren die Lastzeiten über statistische Modelle. [Mar17]

## 2.5 Übersetzungswerkzeuge

In diesem Abschnitt sollen kurz die wichtigsten Grundlagen für alle im weiteren Verlauf verwendeten Übersetzungswerkzeuge und Werkzeugarten dargestellt werden. In diesem Zusammenhang muss kurz auf den Funktionsumfang eines Präprozessor (*Preprocessor*) hingewiesen werden und wesentliche Merkmale eines Übersetzers (*Compiler*) sowie eines Assemblers (*Assembler*) dargestellt werden.

### 2.5.1 Übersetzungsprozess

Wie in Abbildung 2.12 zu sehen ist, gibt es verschiedene Verarbeitungsstufen von einem meist hoch abstrahierten Quellprogramm (*Source Program*), bis zum eigentlichen, auf einer realen Hardware ausführbaren, Maschinencode (*Destination Machine Code*). Die jeweiligen Stufen sind treppenförmig dargestellt, je nach Abstrahierungsgrad und wurden [Aho08] nachempfunden.

Die Aufgabe des Präprozessors (*Preprocessor*) ist in diesem Zusammenhang den zu übersetzenden Quelltext zusammenzustellen und gegebenenfalls Manipulationen daran vorzunehmen. Das bedeutet, ein komplettes Programm aus (Teil-)Programmen zusammen

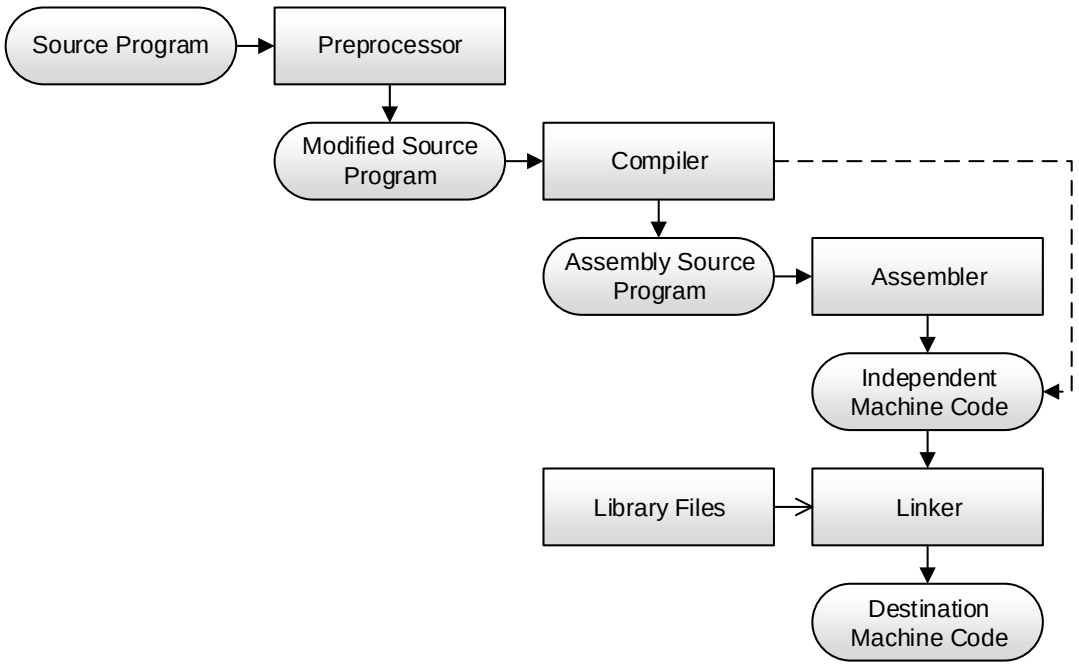



Abbildung 2.12: Allgemeines Sprachverarbeitungssystem

zu setzen und z. B. Makros durch äquivalente Anweisungen zu ersetzen. Das modifizierte Quellprogramm (*Modified Source Program*) kann im Anschluss von einem Compiler übersetzt werden. Ziel eines allgemeinen Compilers ist es, ein Programm in einer Assemblersprache zu erstellen (*Assembly Source Program*). Dieser Zwischenschritt über ein Assemblerprogramm wird gewählt, da dieses leichter auf Fehler zu überprüfen (*Debugging*) ist, als Maschinencode. Es ist aber auch möglich, dass ein Compiler direkt den Maschinencode erzeugt. [Aho08]
Das Assemblerprogramm wird anschließend in den eigentlichen, auf der Hardware ausführbaren Maschinencode übersetzt. Gegebenenfalls ist noch ein Linker (*Linker*) notwendig, der externe Speicheradressen auf physischen Speicher mappt und alle ausführbaren Objektdateien in den Speicher lädt. Dies wird in Abbildung 2.12 als der Schritt von unabhängigen Maschinencode (*Independent Maschine Code*) zum Ziel-Maschinencode (*Destination Machine Code*) dargestellt. [Aho08]

### 2.5.2 Compiler und Assembler

Allgemein betrachtet ist ein Compiler ein Programm, dass einen gegebenen Quellcode einer Quellsprache in eine maschinennahere Form übersetzt. In der Literatur gibt es dabei verschiedene, sich teilweise widersprechende Definitionen. Es wird beispielsweise zwischen Übersetzern im allgemeinen, sowie Compiler und Assemblern als spezielle Übersetzer unterschieden. [Eul13]
In diesem Fall wird in der Literatur unter einem Übersetzer ein Programm verstanden, dass aus einer formalen Quellsprache ein semantisches äquivalent in einer Zielsprache erstellt. Ein Compiler ist in diesem Fall ein spezieller Übersetzer, der diese Übersetzung aus einer Hochsprache in eine Zwischensprache (z. B. Assemblercode) oder Maschinencode realisiert, siehe

Abbildung 2.12.
Ein Assembler ist ein Übersetzer, der ein in Textform gegebenes Programm, der Assemblersprache, in eine eins zu eins Repräsentation in Maschinencode übersetzt. Jedem Maschinenbefehl wird dabei ein Wort (Mnemonic) zugeordnet. Bei einem Assembler wird, im Gegensatz zu einem Compiler, die Logik des zugehörigen Prozessors direkt verwendet und es erfolgt keine Abstraktion der Maschine.
Bei einem Compiler ist im Allgemeinen keine eins zu eins Übersetzung mehr möglich, da die verwendeten Quellsprachen zusätzliche Funktionen, Formulierungsmöglichkeiten von Daten- und Kontrollstrukturen sowie Optimierungen beinhalten. [Eul13, PR84, Kun73]
In Abbildung 2.13 ist die schematische Gliederung der Verarbeitungsschritte eines Compilers gezeigt.

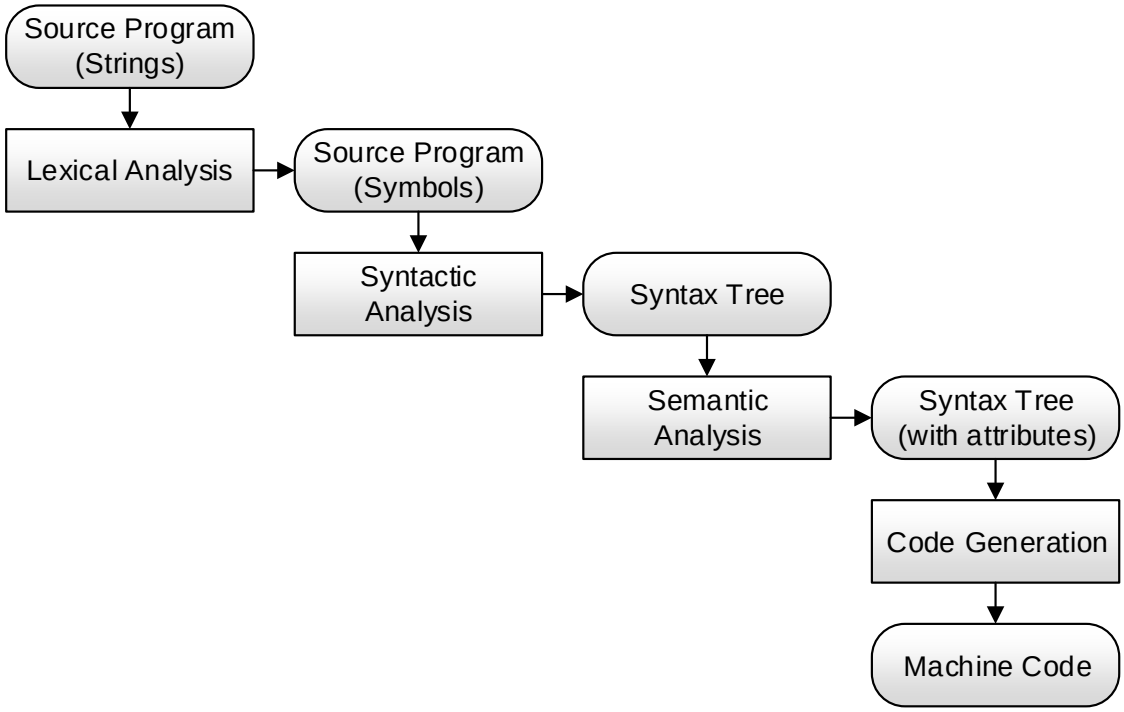



Abbildung 2.13: Gliederung eines Compilers (nach [Eul13])

Diese Verarbeitungsschritte können in einem modular aufgebauten Compiler auch gleichzeitig als austauschbare Module mit definierten Eingabe-/Ausgabe-Verhalten verstanden werden. Zu Beginn erhählt der Compiler ein Quellprogramm in Form einer Zeichenfolge (*Source Program (Strings)*). Auf diesem Quellprogramm wird eine lexikalische Analyse (*Lexical Analysis*) durchgeführt. Das Modul, dass die lexikalische Analyse durchführt, wird häufig als „Scanner“ bezeichnet. Innerhalb des Scanners werden die Zeichen der Eingabe zu Klassen von Symbolen zusammengefasst. Für jedes dieser Symbole, auch Lexeme genannt, wird hier ein Token der Form:

< Tokenname, Attributwert >

ausgegeben. Die damit erzeugt Symbolfolge, bzw. Tokenfolge, (*Source Program (Symbols)*) dient als Eingabe für die Syntaktische Analyse (*Syntactic Analysis*) und wird vom Parser durchgeführt. Hierbei werden die Eingaben auf syntaktische Korrektheit überprüft und etwaige Fehlermeldungen werden generiert. Parallel zu einer Fehlersuche wird ein Syntaxbaum (*Syntax Tree*) erzeugt, bei dem jeder innere Knoten für eine Operation und dessen Kindknoten für die Argumente dieser Operation stehen.
Mit Hilfe dieses Baums werden semantische Analysen (*Semantic Analysis*) durchgeführt. Hierbei werden verschiedene Attribute berechnet, die für den nächsten Schritt der Codeerzeugung notwendig sind. Der semantische Analysator verwendet also den Syntaxbaum (*Syntax Tree*) um das Quellprogramm auf semantische Konsistenz mit der gegebenen Sprachdefinition hin zu überprüfen. Weiterhin werden Typinformationen gesammelt und im Syntaxbaum gespeichert. Diese Typinformationen werden in Abbildung 2.13 als Attribute (*Attributes*) im Punkt *Syntax Tree (with attributes)* bezeichnet. Der letzte Schritt ist die Erzeugung von Assembler- oder Maschinencode (*Machine Code*), je nach Compilerspezifikation, durch einen Codegenerator (*Code Generation*). [Eul13, Aho08]
Für weiterführende Informationen über die detaillierte Codeerzeugung durch Compiler sei auf die einschlägige Literatur verwiesen. Diese Codeerzeugung spielt in dieser Arbeit keine Rolle und deswegen wurde auch nur ein grober Überblick gegeben. Die im Folgenden vorgestellte Speicherverwaltung bei Compilern ist allerdings im weiteren Verlauf der Arbeit von Interesse und wird deswegen genauer betrachtet.
Bei der Codeerzeugung gibt es zwei mögliche Ergebnisse: einen Assemblercode oder einen erzeugten Maschinencode. Nachfolgend soll auf die direkte Erzeugung des Maschinencodes näher eingegangen werden. In diesem Fall werden Register oder Arbeitsspeicherorte für die einzelnen vom Programm verwendeten Variablen ausgewählt. Es ist anzumerken, dass in dieser Arbeit mit dem Begriff Variable die semantische Einheit und unter dem Begriff Speicherzelle der physische Speicherort einer Variable verstanden werden. Anschließend werden die Befehle der letzten Zwischendarstellung in aufgabengleiche Folgen von Maschinenbefehlen übersetzt. Dieses Maschinenprogramm besitzt seinen eigenen logischen Adressraum, in dem jeder Programmwert einen Platz hat. Verwaltung und Aufbau dieses logischen Adressraums werden zwischen Compiler, Betriebssystem und Zielrechner aufgeteilt. Das Betriebssystem bildet dabei den logischen Adressraum auf physische Adressen ab, die über den gesamten Speicher verteilt sein können.
Die Zuweisung von Daten zu physischen Speicherorten (den Speicherzellen) in der Laufzeitumgebung ist der wichtigste Punkt der Speicherverwaltung. Dabei kann die gleiche Variable im Programmcode zur Laufzeit mehrere Speicherzellen bezeichnen. [Aho08]
Bei der Zuweisung von Daten zu physischem Speicher kann es zu Operationsabhängigkeiten, sogenannten Hazards, innerhalb der Instruktionspipeline kommen. Diese Hazards sind Instruktionsreihenfolgeabhängigkeiten und können über entsprechende Mechanismen bei der Speicherverwaltung verhindert werden. Sie treten auf, wenn mindestens zwei Instruktionen $A$ und $B$ auf der selben physischen Speicherzelle operieren, genau dann wenn $A$ in diese Adresse schreibt oder aus dieser Adresse liest und im weiteren Verlauf der Programmabarbeitung $B$ die selbe Zelle schreibt. Diese Hazards können zu unvorhersehbarem Verhalten des Prozessors und schlimmstenfalls zur Zerstörung der Hardware führen. Verhindert werden kann das, indem entsprechende Instruktionen nicht auf der selben physischen Adresse operieren, oder so gescheduled werden, dass diese sich nicht überlagern. Dies geschieht, indem die Lese- und Schreibadressen der jeweiligen Instruktionen miteinander verglichen werden. [CM03]

### 2.5.3 Design Flow bei der Entwicklung mit FPGAs

Der prinzipielle Design Flow von FPGAs ist immer auch von der jeweilig verwendeten FPGA Technologie abhängig. Diese wurde in den vergangenen Jahren kontinuierlich weiterentwickelt. Die wichtigsten Änderungen betreffen vor allem eine erhöhte Hardwaredichte und damit die Integration von immer mehr spezialisierten Komponenten ohne die Anpassung von allgemeinen Ressourcenlayouts und Konfigurationsarchitekturen. [Ebr15]
Geht man von einer allgemeinen Entwicklung nach dem in Abbildung 2.2 vorgestellten V-Modell für eingebettete Systeme aus, dann setzt die Entwicklung auf dem Level der Hardware-Software Partitionierung (*Partitioning*) an. Zunächst wird das geplante Design über in einer Hardwarebeschreibungssprache geschriebene Dateien definiert. Die folgende Abbildung 2.14 ist eine kombinierte Entwurfsmethodik für den Design Flow aus [Ebr15] und [Xil14b].

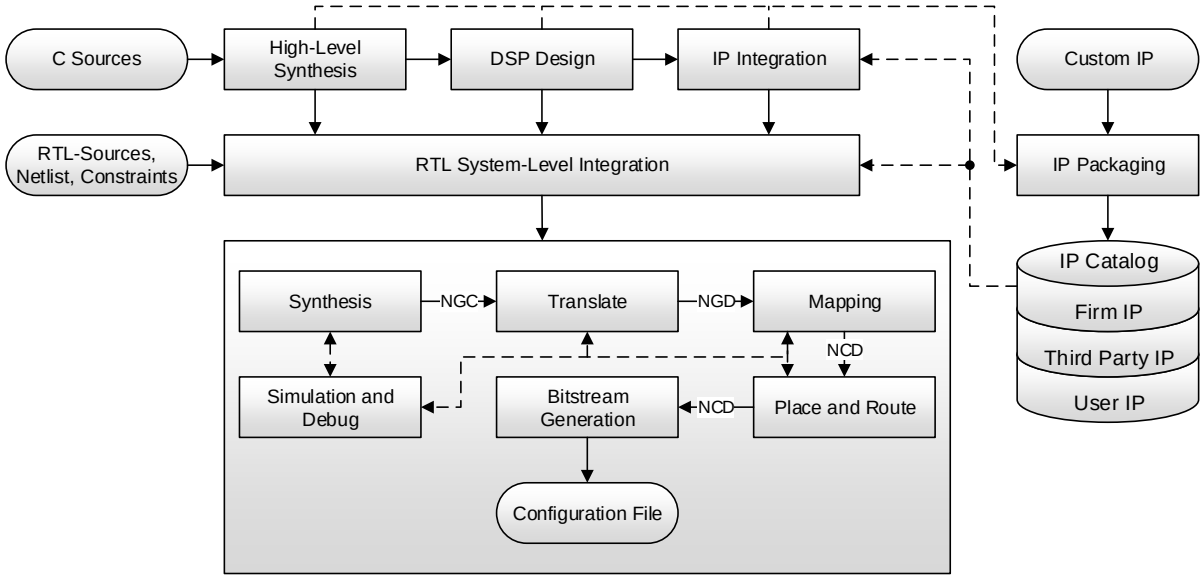



Abbildung 2.14: Allgemeine Entwurfsmethodik für FPGA Designs

Im oberen Teil der Abbildung 2.14 ist der High-Level Design Flow zu sehen. Hier können hoch abstrahierte IP-Pakete (*IP Packaging*) erstellt werden. Diese können in der Hochsprache C geschrieben werden. Über eine entsprechende High-Level Synthese (*High-Level Synthesis*) können diese in den bereits vorhandenen IP-Katalog (*IP Catalog*) integriert werden. Dabei können die IPs aus reiner Logik, eingebetteten Prozessoren, DSP-Modulen oder C-basierten DSP-Algorithmen bestehen. [Xil14b]
Im Stadium der Design Synthese (*Synthesis*) werden die erstellten HDL-Dateien in verschiedene Netzlisten übersetzt. Eine Netzliste wird hierbei als NGC-Datei (*Native Generic Constraint*) bezeichnet.
Im Anschluss erfolgt die Design Implementierung. Dieser Schritt teilt sich in drei Teilschritte: Die Übersetzung (*Translate*), dem Mapping-Prozess (*Mapping*) und dem Platzieren der Logik und physischen Verbinden dieser (*Place and Route*). Im ersten dieser drei Teilschritte, der Übersetzung, werden alle einzelnen NGC-Dateien in eine gemeinsame Datenbasis (*Native Generic Database* (*NGD*)) kombiniert. Diese Datenbasis enthält alle notwendigen low-Level Informationen der benötigten Hardwareressourcen. An dieser Stelle fließen entsprechende nutzerspezifizierte Bedingungen (*Constraints*) in den Prozess ein. Im nächsten Schritt, dem

Mapping, werden der gesamten in der NGD-Datei definierten Logik physische Ressourcen, wie beispielsweise CLBs oder IOBs, zugeteilt. Als Ergebnis dieses Schrittes entsteht die NCD-Datei (*Native Circuit Description*). Im dritten Schritt der Implementierung, dem Place and Route, werden die bereits festgelegten physischen Ressourcen überprüft und eine endgültiges Design und Routing festgelegt.
In einem letzten Schritt wird aus der letzten NCD-Datei ein Bitstream generiert. Dieser repräsentiert die Gerätekonfiguration. [Ebr15]

## 2.6 Ausgewählte Probleme der echtzeitkritischen Daten- und Bildverarbeitung

Echtzeit ist in dieser Arbeit definiert als die Notwendigkeit der Einhaltung von Zeitbedingungen eines Rechensystems an den zugrunde liegenden technischen Prozess. Dabei wird in harte und weiche Echtzeit unterschieden. Die Verletzung der harten Echtzeit führt zu einem nicht akzeptablen, irreparablen materiellen, wirtschaftlichen oder Personenschaden, wohingegen die Verletzung der weichen Echtzeit zu Störungen des Betriebsziels führt, das aber mit Verspätung oder höheren Kosten dennoch erreicht wird.
Echtzeitsysteme werden dabei typischerweise für eine spezielle Anwendung entworfen. Dabei müssen die Architektur, die Fehlertoleranz und gegebenenfalls das Betriebssystem durch zusätzliche Komplexität an die Echtzeitbedingungen angepasst werden. [KS97]
Hierbei entstehen oft gegenläufige Forderungen, die mit dem Ziel der Einhaltung der Echtzeitbedingungen gegeneinander angepasst werden müssen.

Die Klasse der adressierten Echtzeitprobleme, und im speziellen echtzeitkritischen Bildverarbeitungsprobleme, sollen in diesem Kapitel eingegrenzt und näher erläutert werden. Im Folgenden werden die wichtigsten Aspekte der Klasse an verwendeten Algorithmen dargestellt und daraus charakteristische Eigenschaften, die Systeme zur Lösung solcher Probleme besitzen müssen, abgeleitet.

### 2.6.1 Anforderungen der Algorithmen

Die Klasse der adressierten echtzeitkritischen Daten- und Bildverarbeitungsalgorithmen zeichnet sich vor allem durch ihre Anforderung an eine harte Echtzeitschranke aus. Es muss in jedem möglichen eintretenden Fall sichergestellt werden, dass die Verarbeitungseinheit das Ergebnis nach einer definierten Zeitspanne zur Verfügung stellt.
Echtzeitprobleme, im Speziellen alle sich zyklisch wiederholenden Probleme, wie beispielsweise Regelungsprobleme, haben die Anforderung, dass innerhalb eines vorgegebenen Zeitintervalls eine Abfolge von sich zyklisch wiederholenden Operationen abgeschlossen sein muss. In diesem Zusammenhang ist es nicht wichtig, ob diese Abfolge bereits nach einem Bruchteil des Zeitintervalls, oder erst kurz vor der Deadline berechnet ist. Der Aspekt des maximalen Durchsatzes spielt in dieser Algorithmenklasse also nur eine untergeordnete Rolle. Daher ist der wichtigste zu realisierende Aspekt die Eigenschaft nach kompletter Vorhersagbarkeit und Planbarkeit.

Abzuleiten ist die Anforderung der harten Echtzeit bei Bilderverarbeitungen aus dem

praktischen Einsatzgebiet der Verarbeitungseinheiten. Ist das eingebettete System beispielsweise in einer Verarbeitungseinheit integriert, das auf einer Produktionsstraße eingesetzt wird, so muss das Ergebnis, z. B. eine Entscheidung über die Qualität des zu begutachtenden Stücks, innerhalb der Zeit vorliegen, bevor das jeweilige Teil in den nächsten Produktionsschritt gelangt, um gegebenenfalls aussortiert zu werden. Aus der Tatsache, dass Produktionsbänder nicht zeitweise angehalten werden können, weil Teile der Verarbeitung in speziellen Fällen längere Zeiten benötigen, leitet sich der Bedarf nach harten Echtzeitschranken ab.
Weiterhin müssen in Bildverarbeitungsprozessen sehr große Datenmengen verarbeitet werden. Hierbei tritt eine erhöhte Anzahl an Rechenoperationen auf.

Wie bereits erläutert, ist der maximale Durchsatz nicht das primäre Optimierungsziel bei dieser Klasse von Algorithmen. Es kann allerdings als sekundäres Optimierungsziel verstanden werden. Weitere mögliche Optimierungsziele können die Minimierung der verwendeten Chipfläche oder kürzere Latenzzeiten sein. Exemplarisch begründen sich kürzere Latenzzeiten aus der Tatsache, dass wenn bei gleicher Qualität der Lösung eine schnellere Abtastfrequenz möglich ist, ist es entweder möglich innerhalb der gleichen Deadline zusätzliche Aufgaben zu erfüllen, was die Flexibilität für künftige algorithmische Anpassungen steigert, oder die zeitliche Schranke an sich nach unten zu verkürzen, um beispielsweise Produktionsgeschwindigkeiten zu erhöhen. Diese Optimierungsziele sind allerdings in vielen Fällen gegenläufig, so kann der Durchsatz beispielsweise durch eine erweiterte Verwendung von Pipelining und eine Erhöhung von Parallelität gesteigert werden, was gleichzeitig die Latenz vergrößert. Wenn also der Durchsatz einer Teiloperation erhöht wird, wird dadurch die Latenz der Gesamtoperation gesenkt, soll allerdings der globale Durchsatz erhöht werden, so geschieht dies über die Grenzen der Teiloperationen hinweg und resultiert dadurch potenziell in einer erhöhten Latenz der Teiloperation.

Bei Bildverarbeitungsalgorithmen werden nichttriviale Algorithmen, im Sinne der zeitlichen und arithmetischen Anforderungen, benötigt. Die arithmetische Anforderung ist, unter anderem, vor allem die Anforderung an Gleitkomma-Rechenwerke mit hohen Genauigkeiten. Diese Anforderung begründet sich aus den sich zyklisch wiederholenden Operationen mit Rückkopplung. Wird hierbei ein festes Integer-Rechenwerk eingesetzt, so treten aufgrund der Zahlendarstellung mit jeder durchgeführeten Operation kleine Ungenauigkeiten auf, die sich aufgrund der zyklischen Eigenschaft aufaddieren. Weiterhin besitzen die mathematischen Verfahren, die in diesen Algorithmen verwendet werden numerische Eigenschaften, bei denen sich Rundungsfehler akkumulieren können. Ein weiterer Aspekt von Gleitkomma-Rechenwerken ist die Eigenschaft aufgrund der Zahlendarstellung, dass sich die Bitdarstellungen an die jeweiligen Eigenschaften anpassen können und entsprechende Rundungsfehler minimiert werden, da hier die volle arithmetische Auflösung sichergestellt ist.

### 2.6.2 Existierende Lösungen

Aufgrund der Vielfalt an bereits existierenden Lösungsansätzen für die vorgestellten Problemklassen sollen hier exemplarisch nur repräsentative Beispiele vorgestellt werden.

#### Hochpräzise Regelung einer Nano-Positioniermaschine

Ein mögliches Anwendungsszenario ist die hochpräzise Regelung von Stellgrößen. Regler allgemein haben immer die Anforderung nach harter Echtzeit.

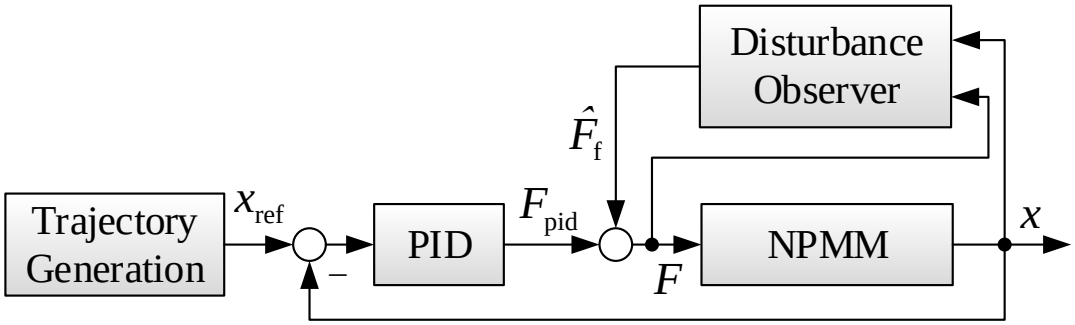



Abbildung 2.15: Kalman Filter und PID-Regler (aus [DPZ+13])

In Abbildung 2.15 ist ein Regelungssystem dargestellt, das die hochpräzise Bewegung einer Nano-Positioniermaschine (*NPMM*) regelt. Für jede Bewegung wird dabei eine dynamische Soll-Wert Vorgabe, also eine Vorgabe der Bewegungskurve (*Trajectory Generation*), erzeugt. Hierbei wird die Reibungskompensation von einem PID-Regler übernommen. Störungen, ausgelöst durch beispielsweise Schallwellen oder Bodenbewegungen werden mittels eines Störungsbeobachters (*Disturbance Observer*) ausgeglichen. Dieser ist als Kalman-Filter realisiert. Innerhalb des PID-Reglers wird über eine entsprechende Feedback-Kontrolle modellbedingte Ungenauigkeiten der Bewegung ausgeglichen. Aufgrund der benötigten sehr hohen Genauigkeit müssen innerhalb der Berechnungseinheiten spezielle Algorithmen verwendet werden, um die reale Geschwindigkeit und die reale Position aus dem potenziell rauschenden Positionssignal zu ermitteln. Dargestellt ist in diesem Fall 1 Kanal, der die Richtung in der X-Achse regelt, eines Drei-Kanal-Systems, die die genaue Position bestimmen. [DPZ+13, ZAM+10]

#### AutoVision SoC Architektur

Ein komplexes Beispiel wird in [Cla11] vorgestellt. Hierbei handelt es sich um ein videobasiertes Fahrer-Assistenz System, dass zur Erhöhung der Sicherheit konzipiert wurde, genannt AutoVision SoC Architektur. Dabei werden von Kameras aufgenommene Videos mit etwa 30 Bildern pro Sekunde verarbeitet. Das bedeutet, dass eine harte Echtzeitanforderung vorliegt, die festlegt das die Verarbeitung eines Bildes in 32.25 ms erfolgen muss. Dieses Fahrer-Assistenz System kann aufgrund der hohen Komplexität und der sich ändernden Anforderungen nicht in reiner Hardware realisiert werden, und eine reine sequentielle Softwareimplementierung mittels Mikrocontrollern führt zu einer Verletzung der Echtzeitbedingungen. Aus diesem Grund wird hier eine Auslagerung performanzlastiger Teile in die Hardware angestrebt, die aufgrund der sich ändernden Anforderungen allerdings flexibel sein muss. Entsprechend wird hier ein FPGA verwendet, der über den Einsatz von verschiedenen Co-Prozessoren ideale Randbedingungen für den Einsatz partieller Rekonfiguration bietet.

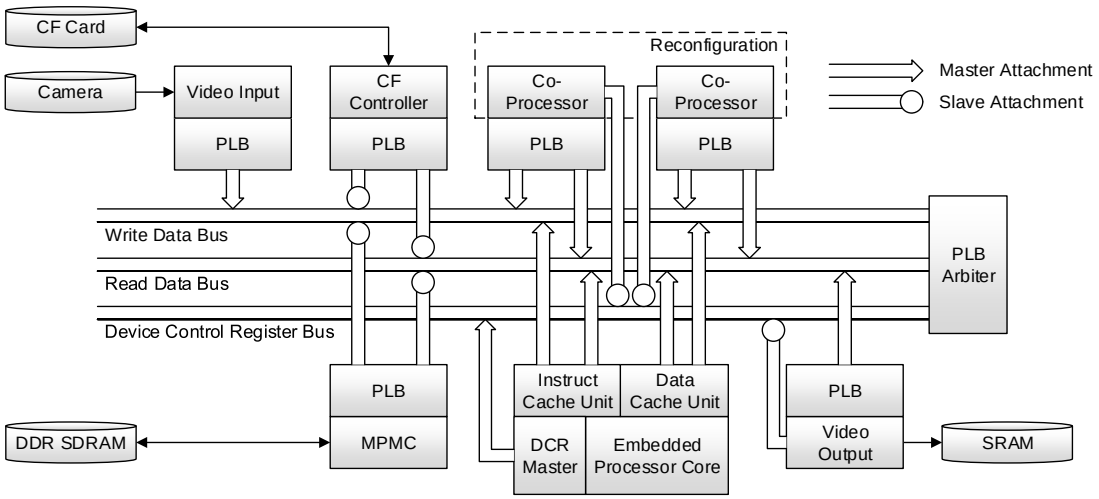



Abbildung 2.16: Blockdiagramm der AutoVision Architektur (nach [Cla11])

Der Datenfluss der Pixeldaten wird mit Abbildung 2.16 verdeutlicht. Dabei werden die Daten von einer Kamera aufgenommen und in einem Video Input Core (*Video Input*) gepuffert, bis sie über den Multi-Port Memory Controller (*MPMC*) in den DDR SDRAM geschrieben werden. Die (Co-)Prozessoren können ebenfalls über den MPMC direkt auf die Eingangsbilddaten zugreifen. Die Kommunikation der meisten Module in der AutoVision SoC Architektur erfolgt mittels sogenannter „Processor Local Bus"-Cores (*PLB*). Diese Module sind für die Übertragung von Daten zwischen beispielsweise CPUs und externen Speicher mit hohen Datenraten gedacht. Am Ende der Datenverarbeitung wird ein Video Output-Core (*Video Output*) die verarbeiteten Daten am Ausgang über einen VGA-Port oder DVI-Port weitergeben.
Bei dieser Architektur wird von der Annahme ausgegangen, dass bei der pixelweisen Verarbeitung von Bildverarbeitungsoperationen oft wiederholende Prozesse auftreten, die mittels einer Hardwareimplementierung über Co-Prozessoren gelöst, und so beschleunigt, werden sollen. Dabei werden alle Elemente der Hardware mit Hilfe einer Modulbibliothek entworfen. Die Elemente dieser Bibliothek lassen sich flexibel zu einer Pixelverarbeitungspipeline zusammenschalten um verschiedene Operationen auf den Pixeln zu realisieren.
Innerhalb der AutoVision Architektur werden also eingebettete Prozessoren (*Embedded Processor Core*) verwendet, deren Funktionalität mittels aufgabenspezifischer Co-Prozessoren (*Co-Processor*) erweitert werden. Diese Co-Prozessoren werden eingesetzt, um verschiedene sich gleichzeitig ausschließende Bildverarbeitungsalgorithmen zu realisieren bzw. zu beschleunigen. Diese Module werden also nach einem Zeit-Multiplex Verfahren gegeneinander ausgetauscht. Als Beispiele von sich gegenseitig ausschließenden Situationen werden Algorithmen zur tageszeitabhängigen Bildverarbeitung, also beispielsweise Nachtlichtsysteme oder Fernlichtassistent genannt, die nur in der Nacht zum Einsatz kommen. Andere Beispiele wären fahrtrichtungsabhängige Funktionalitäten, wie videobasierte Einparkassistenten beim Rückwärtsfahren oder Spurhalteassistenten beim Vorwärtsfahren. Auch wetterabhängige Anpassungen an die Bildverarbeitungen sind möglich.
Diese austauschbaren Co-Prozessoren werden zusätzlich an ein Bussystem angebunden, das Rekonfigurationsinformationen bereitstellt, siehe Abbildung 2.16 der sogenannte Device Control Register Bus (*DCR*).

Die Anforderung an diese Architektur ist, dass pro Sekunde 31 Bilder durch das System verarbeitet werden. Das entspricht einer Bearbeitungszeit pro Bild von 32,25 ms. Das bedeutet, das eine Rekonfiguration des Systems inklusive folgender Bildverarbeitung maximal diese Zeit benötigen darf. [Cla11]

### 2.6.3 Fazit

Die in dieser Arbeit adressierten Problemklassen zeichnen sich vor allem durch die Forderung an harte Echtzeitschranken aus. Dabei werden oft Algorithmen verwendet, die zyklisch wiederholend ablaufen, wie beispielsweise bei Regelungssystemen. Wichtig für die Qualität der realisierten Lösung ist hierbei insbesondere die Rechengenauigkeit durch Verwendung von Gleitkomma-Rechenwerken. Um durch auftretende Rundungsfehler das Messergebnis nicht zu verfälschen ist eine hohe Genauigkeit bei den zu realisierenden Rechenwerken notwendig.
In der vorgestellten Klasse der Bildverarbeitungsalgorithmen ist vor allem eine exakte Vorhersagbarkeit von Verarbeitungszeiten, aufgrund der harten Echtzeitanforderungen, notwendig. Viele bereits existierenden Lösungen bieten diese Planbarkeit zur Designzeit nicht, bzw. haben andere Einschränkungen. Um die in der Bildverarbeitung auftretenden Datenmengen handhaben zu können, sind teilweise Live-Verarbeitungen notwendig, um das zu speichernde Datenvolumen durch Vorverarbeitungen zu minimieren. Die meisten Bildverarbeitungsalgorithmen sind allerdings auf eine Zwischenspeicherung und spätere Verarbeitung der anfallenden Bilddaten angewiesen. Die benötigten Algorithmen für eine Live-Verarbeitung sind nicht trivial und sehr rechenaufwendig.
Die Einschränkung der Lösungsmöglichkeiten aufgrund der kompletten Vorhersagbarkeit, zusammen mit der Verwendung von nicht trivialen Algorithmen, macht die Entwicklung einer speziell dafür angepassten Methodik und praktischen Realisierung notwendig.

## 2.7 Zusammenfassung

In Abschnitt 2.1 wurde definiert, was ein eingebettetes System ist, wie es mit seiner Umwelt interagieren kann und die wichtigsten Anforderungen an solche Systeme wurden genannt. Als Spezialisierung wurde hier bereits auf eingebettete Systeme mit FPGAs eingegangen. Das allgemeine V-Modell für eingebettete Systeme wurde vorgestellt und in den darauf folgenden Kapiteln wurden verschiedene Entwurfsmethodiken diskutiert und markante Beispiele gegeben.

Es wurden die Vor- und Nachteile von sowohl Top-Down, als auch Bottom-Up Systementwürfen in den Unterabschnitten von Kapitel 2.3 erläutert. Als Kernpunkt der notwendigen Grundlagen vermittelt Kapitel 2.4 alle notwendigen Informationen über das Thema FPGA allgemein, die in dieser Arbeit verwendete FPGA-Familie, den speziellen Aufbau als auch die Rekonfigurationsgrundlagen. Es wird allgemein auf SoC-Technologien und im Speziellen auf die SoRC-Technologien eingegangen, sowie die Wichtigkeit und Relevanz von Ressourcenverbrauch und Energieeffizienz bei eingebetteten Systemen mit FPGA-Komponenten herausgearbeitet. Im letzten Abschnitt dieses Kapitels werden Compiler und FPGA Design-Flows diskutiert.
Das letzte Kapitel befasst sich mit den für diese Arbeit relevanten Problemen und Algorithmen der echtzeitkritischen Daten- und Bildverarbeitung. Hier werden wesentliche und notwendige

Eigenschaften vorgestellt sowie bereits existierende Lösungen vorgestellt.
Viele der vorgestellten Eigenschaften haben negative Wechselwirkungen untereinander, so sich kann beispielsweise eine Optimierung des Ressourcenverbrauchs und Senkung des Energiebedarfs negativ auf die benötigte Verarbeitungszeit auswirken. Die Zielstellung bei der Entwicklung von eingebetteten Systemen mit FPGAs muss ein bestmöglicher Kompromiss bei der Gewichtung aller Faktoren, angepasst an die jeweilige Problemstellung, sein.
Um den sich rapide ändernden Charakteristika der eingebetteten Systemen nachzukommen und die Nachteile der vorgestellten Systementwurfsverfahren zu begegnen, sind neue Methodiken notwendig, welche die Vorteile sowohl des Top-Down, als auch Bottom-Up Entwurfs verbinden und dabei den entstehenden Entwurfsoverhead minimieren.

# 3 Entwurf und Anwendung von Logik für Softcore-IPs

In der Vergangenheit wurden rechenaufwendige Probleme häufig mit spezialisierten Prozessorarchitekturen, wie beispielsweise Graphic Processing Units (*GPUs*) oder DSPs, gelöst. Die dabei realisierte Hardwareimplementierung kann aufgrund der Spezialisierung nur sehr schlecht für andere Probleme wiederverwendet werden. Die Realisierung mit einer GPU ist zwar bereits flexibler, allerdings haben auch diese Nachteile, beispielsweise ihre hohe Leistungsaufnahme, was sie für viele Einsatzgebiete, gerade mit begrenzten Energiereserven, ungeeignet macht. [TC12]
Mit steigendem Marktanteil und der Leistungssteigerung von FPGAs, erschließt sich eine weitere Alternative. Innerhalb von FPGAs können über die Kombination verschiedener IPs Schaltungen realisiert werden. So ist es auch möglich, ganze Prozessoren funktionell nachzubauen. Haben diese Prozessoren mehrere rekonfigurierbare Ausführungseinheiten und besitzt die verwendete Befehlssatzarchitektur die Eigenschaft „Very Long Instruction Word (*VLIW*)“, dann werden diese in FPGAs umgesetzten Prozessoren als Softcore Prozessoren, oder kurz Softcores, bezeichnet. Softcores sind damit „weich“ realisierte Elemente, das bedeutet, dass sie komplett über logische Primitive auf dem FPGA implementiert werden. [WAB08, IS93]
VLIW bedeutet hierbei, dass die auszuführenden Befehle nicht dynamisch zur Laufzeit vom Prozessor den einzelnen Ausführungseinheiten zugewiesen werden, sondern der Compiler (oder der Assembler) zur Designzeit eine parallel ausführbare Befehlsabfolge erzeugt. Das Ziel einer Befehlssatzarchitektur mit dieser Eigenschaft ist es, die Abarbeitung von sequentiellen Programmen durch Ausnutzung von Parallelität auf Befehlsebene zu beschleunigen. [Mat01]
In einem idealisierten Entwicklungsprozess werden zu Beginn alle Anforderungen festgelegt und ändern sich im Projektverlauf nicht. In der Praxis gibt es allerdings im Verlauf eines Projektes oft Änderungen der Anforderungen, die zu einem Wechsel, einer Veränderung oder einem Hinzufügen von zusätzlichen Teilen im Design führen. Die Verwendung von Softcore Prozessoren minimiert oder eliminiert die dadurch entstehenden Probleme aufgrund der großen Flexibilität. [AE06]
Softcore-IP steht dabei für Softcore-Intellectual Property, also Softcore Prozessoren. Es werden zwei Arten von Softcore Prozessoren unterschieden werden:

- universelle Softcore Prozessoren (z. B. der Leon3 Softcore [Cob17])
- applikationsspezifische Softcore Prozessoren

Der wesentliche Unterschied liegt im Einsatzgebiet der Prozessoren. Universelle Softcores sind dafür entwickelt, ein möglichst breites Aufgabenspektrum abzudecken und damit maximale universelle Verwendbarkeit zu besitzen. Wohingegen applikationsspezifische Softcores ein kleineres Einsatzgebiet bedienen, dafür aber speziell auf die dortigen Problemstellungen hin optimiert sind und damit eine bestmögliche Erfüllung aller gegebenen Anforderungen erreichen. Die Anforderungen können dabei auch gegenläufig sein, wie beispielsweise Energieverbrauch

und maximaler Durchsatz, sodass alle Anforderungen zwar erfüllt werden, aber meist nicht übererfüllt.

## 3.1 Gesamtentwurf

Softcore Prozessoren sind aufgrund ihrer Implementierung sehr flexibel. So kann der Einsatzbereich eines Softcore von einfachen Eingabe-zu-Ausgabe Manipulationen bis hin zu komplexen Systemen reichen.

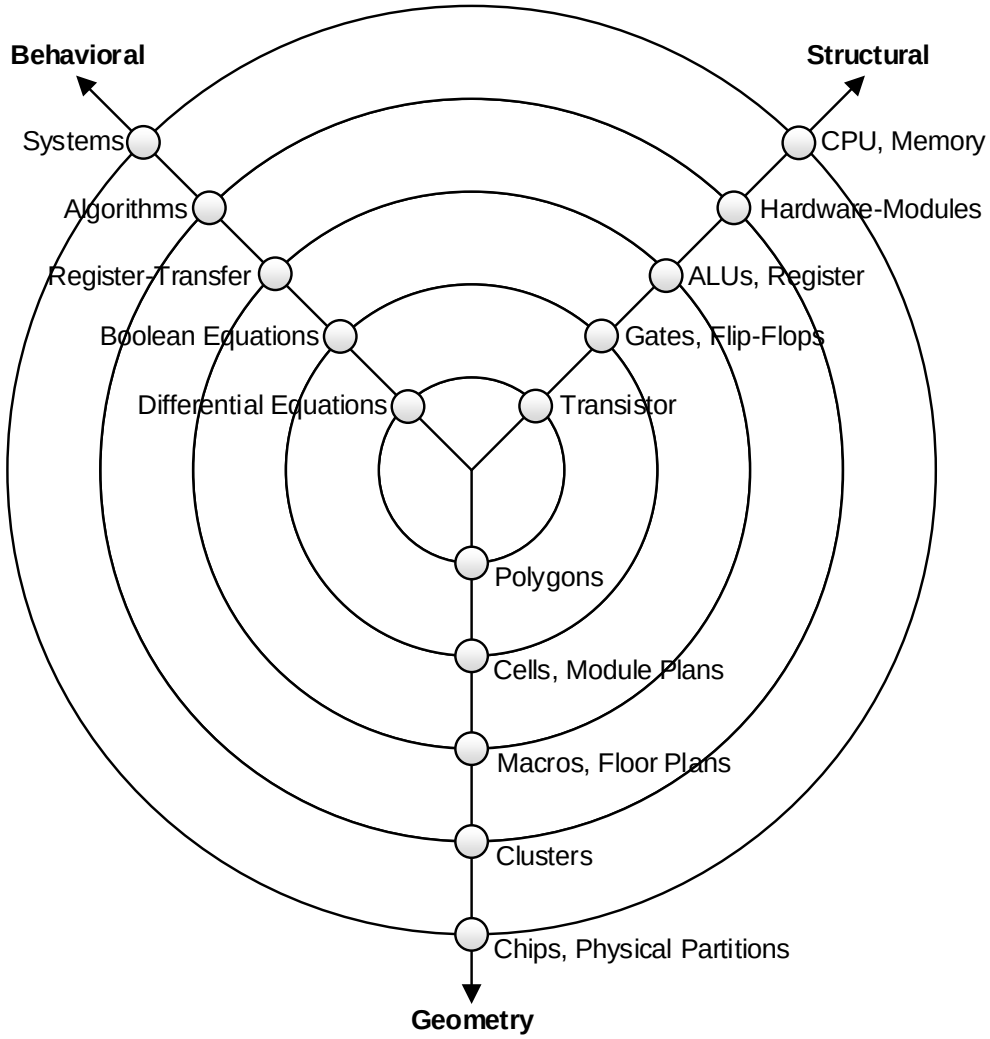



Abbildung 3.1: Y-Diagramm (nach Gajski-Walker-Kuhn)

Um einen Gesamtsystementwurf eines Softcore Prozessors sinnvoll modellieren zu können, ist das Y-Modell nach Gajski-Walker-Kuhn aus Abbildung 3.1 in abgewandelter Form gut geeignet. Ursprünglich dient das Modell dazu, bei dem Design von integrierten Schaltkreisen mit der

Komplexität von „Very Large Scale Integration (*VLSI*)“, also einer Komplexität von 100.000 Transistoren pro Flächeneinheit oder darüber, verwendet zu werden. Im Folgenden wird zunächst das allgemeine Y-Diagramm erläutert und anschließend speziell auf die Softcoreentwicklung angepasst. [TG06]
In der in Abbildung 3.1 gezeigten Form wird das Modell für die Schaltungsentwicklung von ASICs mit mindestens 100.000 Transistoren pro Flächeneinheit verwendet. Wie in Abbildung 3.1 dargestellt, kann der gesamte Entwurf auf verschiedene Modellierungsebenen verteilt werden. Das Modell lässt sich dabei in drei Domänen einteilen: die Verhaltensdomäne (*Behavioral*), die Strukturdomäne (*Structural*) und die Geometriedomäne (*Geometry*). Entlang der dargestellten drei Achsen erfolgt dabei von außen nach innen eine Entwurfsverfeinerung. Die Gegenrichtung kann als eine Abstraktion des Systemverhaltens verstanden werden. Die kreisförmigen Verbindungen zwischen den drei Achsen werden als Hierarchieebenen verstanden. [Wec15]
In der Verhaltensdomäne (*Behavioral*) wird das allgemeine Systemverhalten beschrieben. Diese Beschreibung erfolgt mit Ablaufplänen, Zeitdiagrammen und Kennwerten aus Datenblättern und wird schrittweise verfeinert. Hierbei muss darauf geachtet werden, dass der Abstraktionsgrad der verwendeten Modelle auf die aktuelle Entwurfsebene angepasst ist. In der Ebene des Systems (*Systems*) werden deswegen häufig Registertransferbeschreibung, also beispielsweise Datenflüsse zwischen den einzelnen Logikfunktionen und auf der algorithmischen Ebene (*Algorithms*) Hardwarebeschreibungssprachen wie VHDL oder Verilog, verwendet. In den unteren Ebenen wird das Verhalten durch Simulatoren und Analyseprogramme, wie Circuit- oder Logiksimulatoren, ermittelt. [Sti16]
In der Strukturdomäne (*Structural*) wird der Aufbau des Systems über Komponenten beschrieben. Auf der obersten Ebene stehen dabei die komplexesten Komponenten, wie beispielsweise ganze Prozessoren (*CPU*), oder Speichersysteme (*Memory*). Je weiter unten die Ebenen angeordnet sind, je einfacher sind die verwendeten Komponenten. So vereinfachen sich die Komponenten beispielsweise von kompletten Prozessoren, über Addierer oder Register bis hin zu einfachen Transistoren oder Flip-Flops (*FFs*). Die Schaltungen der jeweiligen Komponenten werden dabei in Form von logischen Schaltplänen, Blockschaltbildern oder Netzlisten beschrieben. [Sti16]
Innerhalb der Geometriedomäne (*Geometry*) wird die Anordnung der einzelnen Bauteile spezifiziert. Die Geometrie beschreibt dabei die physikalische Implementierung einer Schaltung auf dem Chip. Hierbei kommen Layout-Editoren, Platzierungs- und Routing-Werkzeuge sowie Design-Rule-Checker zum Einsatz.
Domänenübergreifend werden fünf Entwurfsebenen unterschieden, wie in Abbildung 3.2 dargestellt.

Auf der Systemebene (*System Level*) wird das Verhalten der Schaltung (Verhaltensdomäne) beschrieben, welche Bauelemente diese enthält (Strukturdomäne) und die physische Realisierung (Geometriedomäne). Die algorithmische Ebene (*Algorithm Level*) gibt die Art der Realisierung durch definierte Funktionen wieder. Die dritte Ebene ist die funktionale Ebene (*Functional Level*). Sie wird auch als Register-Transfer Ebene bezeichnet. An dieser Stelle wird das Verhalten anhand der Transferoperationen der einzelnen Daten zwischen den Registern beschrieben. Die Logikebene (*Logic Level*) beschreibt das Verhalten anhand einzelner Logikgatter und die Schaltkreisebene (*Circuit Level*) stellt den Entwurf der Schaltung mit elektronischen Grundbausteinen, wie Transistoren oder Widerständen dar. [Sti16]

Soll das Modell speziell auf die Entwicklung eines Softcore und damit für die Entwicklung mit FPGAs angewendet werden, so sind entsprechende Spezifikationen notwendig. Es müssen dabei lediglich vier Modellierungsebenen durchlaufen werden, von der Systemebene (*Systems*) bis zur

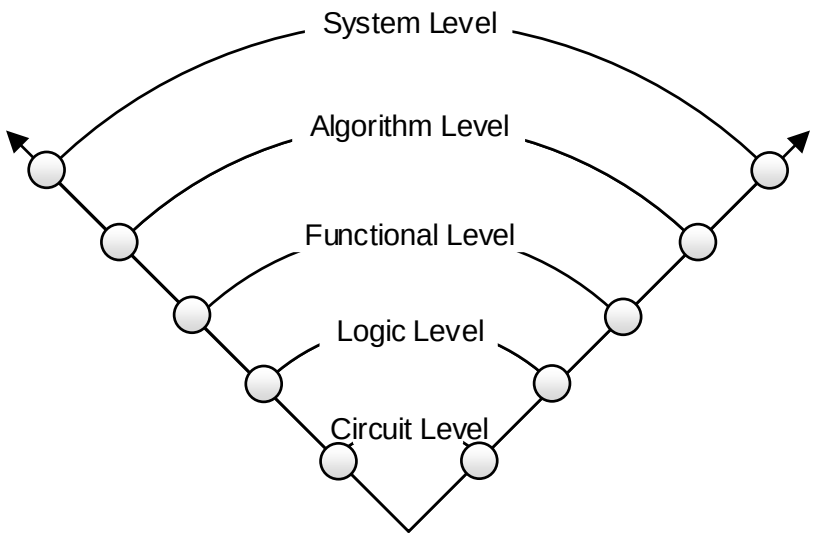



Abbildung 3.2: 5 Ebenen im Y-Diagramm (nach [Sti16])

Logikebene (*Logic*). Die unterste im Y-Modell dargestellte Ebene, die Schaltkreisebene (*Transfer Functions*), wird bei einem Entwurf mit FPGAs nicht benötigt, da der FPGA bereits fertige Gatterstrukturen besitzt und diese verwendet werden. Betrachtet man die Geometriedomäne (*Geometry*) des Diagramms, so ist erkennbar, dass bei der Entwicklung mit FPGAs die beiden untersten Ebenen, sowohl die Polygone (*Polygons*), als auch die Zellen (*Cells*), nicht entwickelt werden müssen, da diese bei einem FPGA bereits realisiert vorliegen. Verdeutlicht wird das in Abbildung 3.3.

Auf der Systemebene (*Systems*) werden alle vorhanden Spezifikationen an das Gesamtsystem zusammengetragen. Bei einem Softcore ist das vor allem das zugrunde liegende Grundmodell, das Befehlsformat und die I/O-Möglichkeiten. Das Grundmodell enthält dabei Informationen über:

- Befehlssatz
- Datenformat
- Adressierung
- Architektur

An dieser Stelle wird eine weitere wichtige Anpassung deutlich: Im unveränderten Y-Diagramm enthält das Grundmodell keine Angaben über die Architektur, dafür aber Informationen über die spezifizierten Taktfrequenzen des gesamten Systems. Diese werden allerdings bereits von dem fertigen FPGA vorgegeben und müssen deswegen nicht mehr betrachtet werden. Sollten innerhalb der Architektur allerdings verschiedene Taktdomänen geplant sein, müssen diese als Taktkonzept hier ebenfalls betrachtet werden.
Auf der algorithmischen Ebene (*Algorithms*) werden die Algorithmen spezifiziert, die in den Subsystemen ablaufen sollen. Dies können beispielsweise Informationen über die Größe von Daten-, Adress-, und Befehlsformaten sein, welche Zahlenformate verarbeitet werden müssen

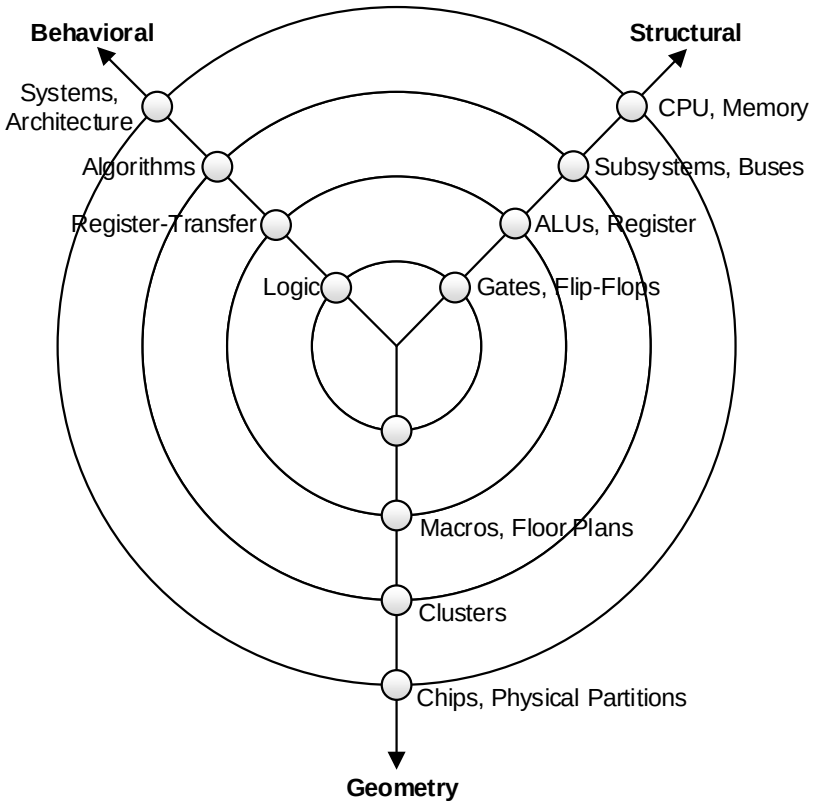



Abbildung 3.3: Y-Diagramm für FPGA Entwicklungen

und der zugrunde liegende Befehlssatz (sowohl arithmetische, als auch logische Befehle). Zusätzlich wird die Adressierung festgelegt.
Auf der Register-Transfer-Ebene (*Register-Transfer*) werden Zeitabhängigkeiten zwischen den einzelnen Komponenten definiert. Das Taktschema für alle verwendeten Register wird festgelegt und eine etwaige Reihenfolge der Operationen wird definiert. An dieser Stelle kann definiert werden, dass Maschinenbefehle weiter in Mikrooperationen aufgeteilt werden. Die Reihenfolge der Mikrooperationen wird dabei durch den einzelnen Befehl bestimmt und es muss sichergestellt werden, das die Teilergebnisse rechtzeitig in den dafür vorgesehenen Registern anliegen.
Auf Logikebene (*Logic*) wird das Verhalten über Boolesche Gleichungen beschrieben. Auf dieser Ebene befinden sich die Informationen über das exakte Zeitverhalten der einzelnen Gatter. [Wec15]

Auf der obersten Ebene der Strukturdomäne (*Structural*) wird eine Systemebeneneinteilung der Komponenten vorgenommen. Es können CPU, Speicher und Bussysteme definiert werden. Für diese Komponenten müssen Systemparameter definiert werden. Die Strukturierung auf dieser Ebene ist dabei unabhängig von der Zielhardware.
Die Aufteilung der auf der nächsten Ebene befindlichen Subsysteme (*Subsystems, Buses*) ist ein entscheidender Schritt in der Entwicklung eines Softcore, da von diesem Entwurfsschritt der gesamte weitere Entwurf beeinflusst wird. Es kann eine Einteilung in folgende exemplarische Subsysteme vorgenommen werden:

- Operationswerk (Datenpfad)
- Steuerwerk (Steuerpfad)
- Ein- und Ausgabeeinheit
- Bussystem

Auf dieser Ebene erfolgt ebenfalls die Zuordnung der Algorithmen zu den strukturierten Komponenten. Meist erfolgt aufgrund der großen Komplexität eine Aufteilung der Subsysteme nach Datenpfad und Steuerpfad.
Die Modellierung von einzelnen Registern und der ALU erfolgt auf dem Register-Transfer-Level. Hierbei wird mindestens ein Ergebnisregister verwendet. Leitungen und Busse werden definiert und der Datentransfer zwischen den einzelnen Komponenten erfolgt über Multiplexer, Demultiplexer und Tri-State-Treiber. [Wec15]

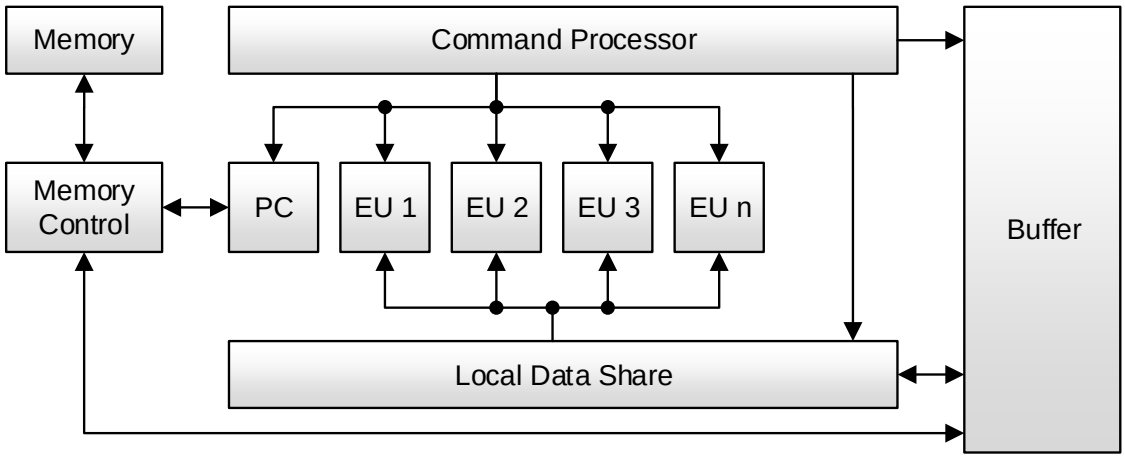



Abbildung 3.4: Exemplarischer Aufbau eines Softcore (angelehnt an [TC12])

In Abbildung 3.4 ist ein exemplarischer Softcore dargestellt. Anhand dieses relativ generischen Beispiels lassen sich die im weiteren Verlauf der Dissertation entwickelten wichtigen Ideen und Umsetzungen gut exemplarisch erläutern. Dieser Softcore besitzt eine 32 Bit Datenbreite, bei einer 10 Bit Adressbreite und verfügt über eine Ausführungseinheit, die nicht gepipelined ist. Er ist dem Single Instruction Multiple Data (*SIMD*) Modell einer GPU nachempfunden. Das bedeutet, dass das Parallelitätslevel von der Anzahl an Verarbeitungseinheiten (*Execution Units* (*EU*)) abhängig ist. Diese ist innerhalb des Softcore das zentrale Verarbeitungselement und beinhaltet arithmetische und logische Operationen. Es besteht dabei jeweils aus einem Array aus allgemeinen Registern, einem Ergebnisregister und einem Akkumulatorregister.
Der Kommandoprozessor (*Command Processor*) speichert die Befehle aus dem Systemspeicher. Dabei bestimmt der Operand der Instruktion die allgemeinen Register innerhalb der Verarbeitungseinheiten. Weiterhin spezifiziert dieser über die Instruktionen die Adresse der Module, die die Berechnungen ausführen sollen.
Der Speicherkontroller (*Memory Control*) verbindet den lokalen Speicher mit dem Systemspeicher und der Programmzähler (*PC*) ermittelt die aktuelle Position im Programm. Der lokale Datenspeicher (*Local Data Share*) sendet und empfängt die Daten aus den Verarbeitungseinheiten. Diese können über den lokalen Speicher ihre Daten mit anderen

Verarbeitungseinheiten austauschen. [TC12]

Es gibt verschiedene Methoden, wie die Peripherie an den Softcore angeschlossen werden kann, was ein weiterer Vorteil ist, der sich aus der großen Flexibilität ergibt. Das SoRC umfasst dabei speicherbasierte anschließbare Geräte mit Lese-/Schreib-Interfaces. Hierbei sollte jedes Peripheriegerät innerhalb eines SoRC das gleiche Interface benutzen, um die Möglichkeit der Verwendung einer Busstruktur zu schaffen. Der Vorteil einer solchen Busstruktur liegt darin, dass ein vorgefertigtes Tool die Busstruktur erstellen und verarbeiten kann. Ein weiterer Vorteil ist, dass es bei einem standardisierten Interface nicht notwendig ist, Informationen über den internen Aufbau eines Moduls zu haben, um es verwenden zu können. Da FPGAs in SoRCs zum Einsatz kommen, ist auch die I/O-Pinbelegung flexibel. Es ist möglich diese komplett an die jeweilige Problematik anzupassen und dennoch nach dem Platzieren der Komponenten allen zeitlichen Anforderungen gerecht zu werden. Zusätzlich zu den frei wählbaren Input-/Output-Pinbelegungen verfügen FPGAs über mehr General Purpose Input/Output (*GPIO*). Es kann, je nach Bedarf, ein größerer FPGA mit weiteren GPIOs verwendet werden, sollte es die Applikation erfordern. Die Flexibilität ist damit wesentlich größer als bei beispielsweise einem ASIC.
Ein Softcore kann damit auf sehr viele Probleme speziell zugeschnitten werden. Das kann, abhängig von dem speziellen Prozessor, auf verschiedenen Ebenen geschehen. Entsprechende Tools können Anpassungsmöglichkeiten, wie beispielsweise die Cachegröße, die Anzahl an Pipelinestufen oder den maximal verwendbaren Speicher bieten. Hierbei ist zu beachten, dass eine Optimierung, die eine Steigerung der Performanz bewirkt, auch immer eine erhöhte Verwendung von logischen Elementen zu Folge haben wird. Es können bereits vor Beginn einer neuen Entwicklung verschiedene Versionen eines Prozessorkerns vorliegen, die verschiedenen Performanzspezifikationen entsprechen, aus denen gewählt werden kann. In einer tieferen Entwicklungsebene ist es dann möglich, das Entwickler direkt Modifikationen am Quellcode vornehmen, um Projektvorgaben gerecht zu werden. [AE06]

#### Design- und Verifikationszyklen

Ein wichtiger Kostenfaktor bei der Entwicklung von SoCs ist die Zeit, die ein Design Zyklus benötigt. Wie bereits erörtert, bietet ein Softcore Prozessor dabei die Möglichkeit von sehr kurzen bzw. schnellen Design- und Verifikationszyklen. So ist es möglich, dass im Softcore verwendete Software bereits in einem sehr frühen Entwicklungsstadium getestet werden kann. Es besteht nicht die Notwendigkeit, dass der Softwareentwickler auf die Fertigstellung des Boards warten muss, um die Software testen zu können. Weiterhin kann ein Softcore-System, je nach Zielapplikation, sehr schnell aufgesetzt werden. Es gibt spezielle herstellerspezifische Entwicklungswerkzeuge, die genau für diesen Einsatzzweck entwickelt wurden und verwendet werden können. Die Verwendung von standardisierten vorgefertigten Schnittstellen verringert die Entwicklungszeit zusätzlich. Die Entwicklung der Hardware ist nicht auf das Herstellen von physischen Prototypen angewiesen, was die Zeit der Designzyklen stark verkürzt. Eine weitere Möglichkeit, die Entwicklungszeit zu verkürzen ist die Hardware-Software Co-Design und wird in Kapitel 3.4 genauer beschrieben. [AE06]

## 3.2 Entwurf von Teil-IP Logik

Bei der Entwicklung von IP Logik für Softcore Prozessoren sollten grundsätzlich zwei Fälle unterschieden werden:
Im ersten Fall wird lediglich Logik, wie im Kapitel 2.3.2 vorgestellt, als allgemeines Bottom-Up Verfahren entwickelt. Hierbei liegt das Augenmerk auf der Erweiterung möglicher Funktionalitäten ohne spezifische Applikationsanforderung, oder für einen umzusetzenden Algorithmus, also mit spezifischer Applikationsanforderung.
Im zweiten Fall wird die Logik mit dem Hintergrund der partiellen Rekonfigurierbarkeit (Partial Reconfiguration (PR)) entwickelt. Hierbei muss auf eine Reihe von Eigenschaften geachtet werden, die im folgenden erläutert werden.

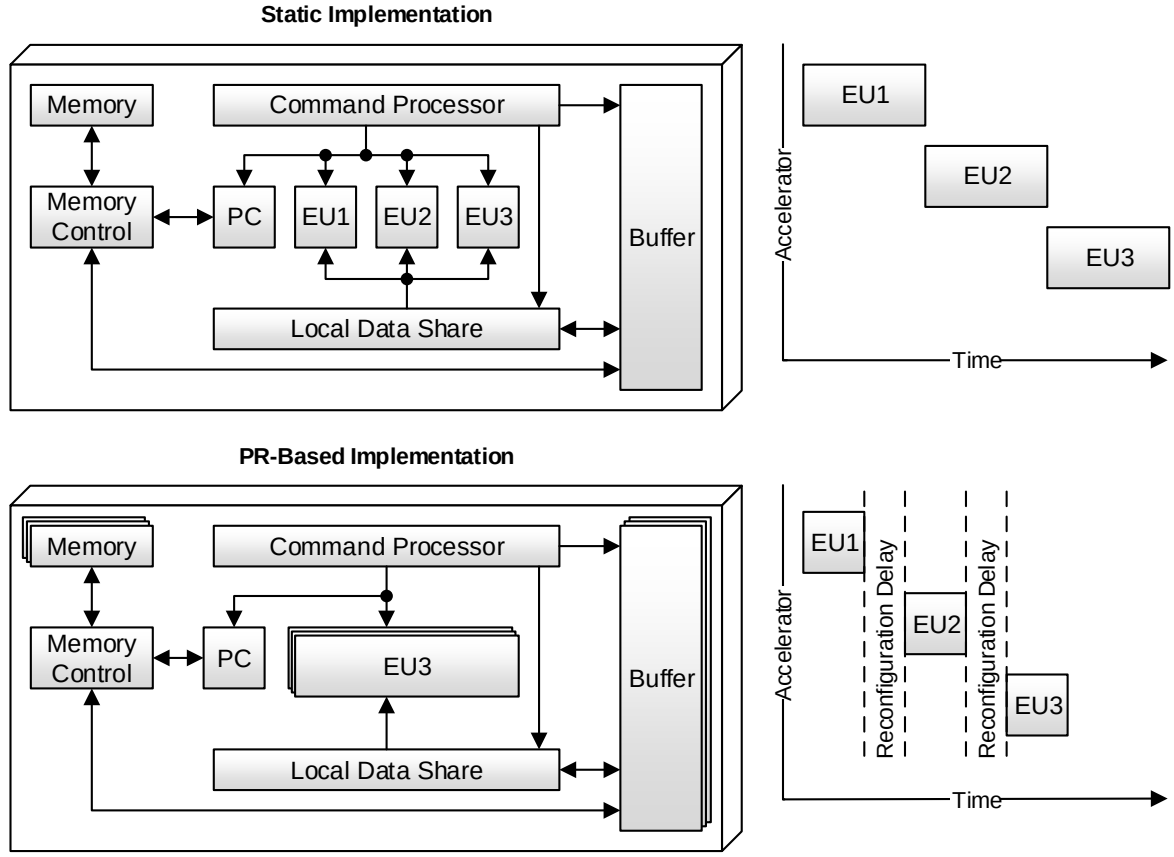



Abbildung 3.5: Statische Implementierung vs. PR-basierte Implementierung

In einem ersten Entwurfsschritt muss entschieden werden, welche Teile des Designs ausgetauscht werden sollen. Hierbei gibt es, je nach Architektur und Anwendungsszenario, verschiedene in Abbildung 3.5 dargestellte Möglichkeiten. Ein Anwendungsfall ist der Austausch des zu rechnenden Algorithmus bei ansonsten gleichbleibenden Verarbeitungseinheiten (*EU*). In diesem Fall würde der Speicher (*Memory*) und die Startbelegungen der Speicherzellen (*Buffer*) ausgetauscht werden. Ein anderer Anwendungsfall kann das Austauschen von Verarbeitungseinheiten (*EU*) sein. Dieser Fall tritt vor allem dann auf, wenn über einen definierten

Zeitraum nur wenige Verarbeitungseinheiten benötigt werden. Hier besteht die Möglichkeit, den gesamten zur Verfügung stehenden Platz einer Verarbeitungseinheit zuzuweisen und dafür die Verarbeitungsgeschwindigkeit dieser Einheit zu maximieren. Sobald diese Einheit nicht mehr benötigt wird, wird sie durch die nächste Einheit ersetzt. Dieser kann nun ebenfalls die maximalen Ressourcen zugeordnet werden, und damit die Geschwindigkeit der Berechnung maximiert werden. Hierbei ist zu beachten, dass der Austausch der Module ebenfalls Zeit benötigt, die nicht zur Berechnung des Algorithmus verwendet werden kann.
Eine Kombination, also ein Wechsel des Algorithmus und parallel eine Anpassung der Verarbeitungseinheiten ist, je nach Anwendungsfall und Architektur des Softcore, ebenfalls möglich.

## 3.3 Entwurfsschritte beim Entwurf von Softcore-IP Logik

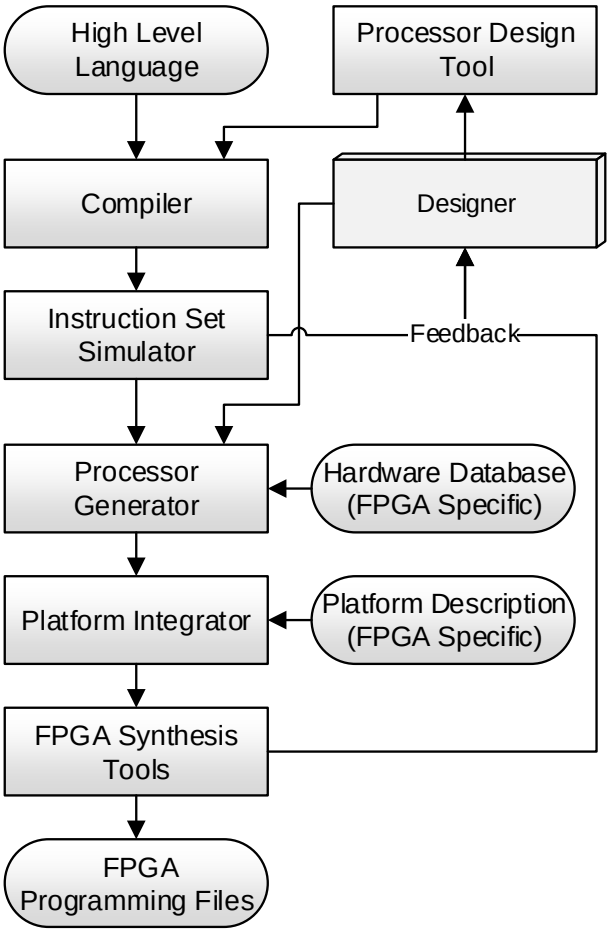



Abbildung 3.6: Synthese eines Softcore (nach [EJH+10])

Ein allgemeiner Design Flow wird in Abbildung 3.6 vorgestellt. Dieser Design Flow ist eine praktische Implementierung des in Kapitel 2.1 unter Abbildung 2.2 vorgestellten V-Modells und setzt ab dem Punkt der Partitionierung (*Partitioning*) auf der linken Seite des Modells

an. Hierbei wird davon ausgegangen, dass die Beschreibung des zu erstellenden Softcore Prozessors in einer Hochsprache (*High Level Language* (*HLL*)), beispielsweise C,C++ oder OpenCL, vorliegt. Diese als initiale Architektur bezeichnete Grundversion soll dabei nur ein minimales Set an Ressourcen, also Verarbeitungseinheiten, verwenden und über eine generische Architektur aus der HLL erstellbar sein. Aus dem HLL Quellcode wird in einem ersten Schritt über einen Compiler das Assembler-Code Programm, also die Anweisungen für den Softcore, erstellt. Mit Hilfe der Ergebnisse eines Simulators (*Instruction Set Simulator*) kann ein Designer entsprechende Anpassungen am geplanten Softcore und dem Assembler-Code vornehmen. Übliche Ergebnisse des Simulators sind dabei Angaben über Ressourcenverbrauch und benötigte Rechenzeiten. Hierbei kann das Instruction Set, je nach verwendetem Softcore, entweder fest vorgegeben, oder variabel anpassbar sein. Alle vorgenommenen Anpassungen müssen als Verarbeitungseinheiten in der Datenbank (*Hardware Database*) für den zu verwendenden FPGA vorliegen. Der Prozessor Generator wird dann die verwendeten Verarbeitungseinheiten aus der Hardwarebibliothek laden, eine Kontrolleinheit generieren und ein entsprechendes Verbindungsnetzwerk. Die Kontrolleinheit beinhaltet dabei den Instruction Decoder, optionale Dekompressionseinheiten und eine Befehls-Fetcheinheit. Anschließend ist der Plattformintegrator (*Platform Integrator*) dafür zuständig den Speicher und externe IPs mit dem Prozessor zu verbinden. Weiterhin werden alle notwendigen Projektdateien für dritte Synthesetools generiert und das physische I/O-Mapping festgelegt. [EJH+10]
Der letzte Schritt der Synthese, Mapping und Place and Route erfolgt über externe Tools, wie in Abbildung 2.14 unter Kapitel 2.5 beschrieben.

## 3.4 Hardware-Software Co-Design

Das übergeorndete Ziel beim Co-Design von Hard- und Software ist die Senkung von Entwicklungszeit und damit auch Entwicklungskosten. Da die Komplexität moderner Systeme keine direkte Hardwareumsetzung mehr zulässt, ist es notwendig Modelle zu verwenden, die eine derivative Umsetzung ermöglichen. Das primäre Ziel dabei ist eine möglichst späte Partitionierung zwischen Hard- und Software vorzunehmen, um mehr Freiheitsgrade bei den Optimierungen zu generieren. Ein wichtiger Aspekt hierbei ist die Möglichkeit, verfeinerte Tests auszuführen, bevor sich auf spezielle Hard- und Softwarekomponenten festgelegt werden muss. Ein weiterer Vorteil dieser Designmethodik liegt in den kürzeren Rückschleifen im Falle einer notwendigen Änderung von einzelnen Komponenten. Hierbei ermöglichen die unterschiedlichen Abstraktionslevel eine Separierung von Funktion, Architektur und Realisierung. Die verschiedenen Abstraktionslevel für das Co-Design von Hardware und Software sind in Abbildung 3.7 dargestellt.

Als Designeinstieg werden auch hier entsprechende Spezifikationen (*Specification*) benötigt. Es werden Informationen über die Umgebung (*Environment*) des einzubettenden Systems benötigt, ebenso wie die Anforderungen (*Requirements*) und Einschränkungen (*Constraints*). Zusätzlich müssen hier Informationen über eine Testbench frühzeitig in der Entwicklung festgelegt werden. Diese Anforderungen definieren die System Architektur (*System Architecture*). Auf dieser Ebene wird direkt die Trennung in Software- und Hardware-Modell vorgenommen. Auf diesem Level ist eine Toolgestützte Synthese (*High Level Synthesis*) beispielsweise mit SystemC oder SystemVerilog denkbar. Aufgrund des Abstraktionsgrades gibt es an diesen Stellen noch keine zeitlichen Aussagen, es ist also „Untimed“. Jetzt wird in einer parallel laufenden Co-Entwicklung sowohl Hardware, als auch benötigte Software entwickelt. In den jeweiligen Stufen

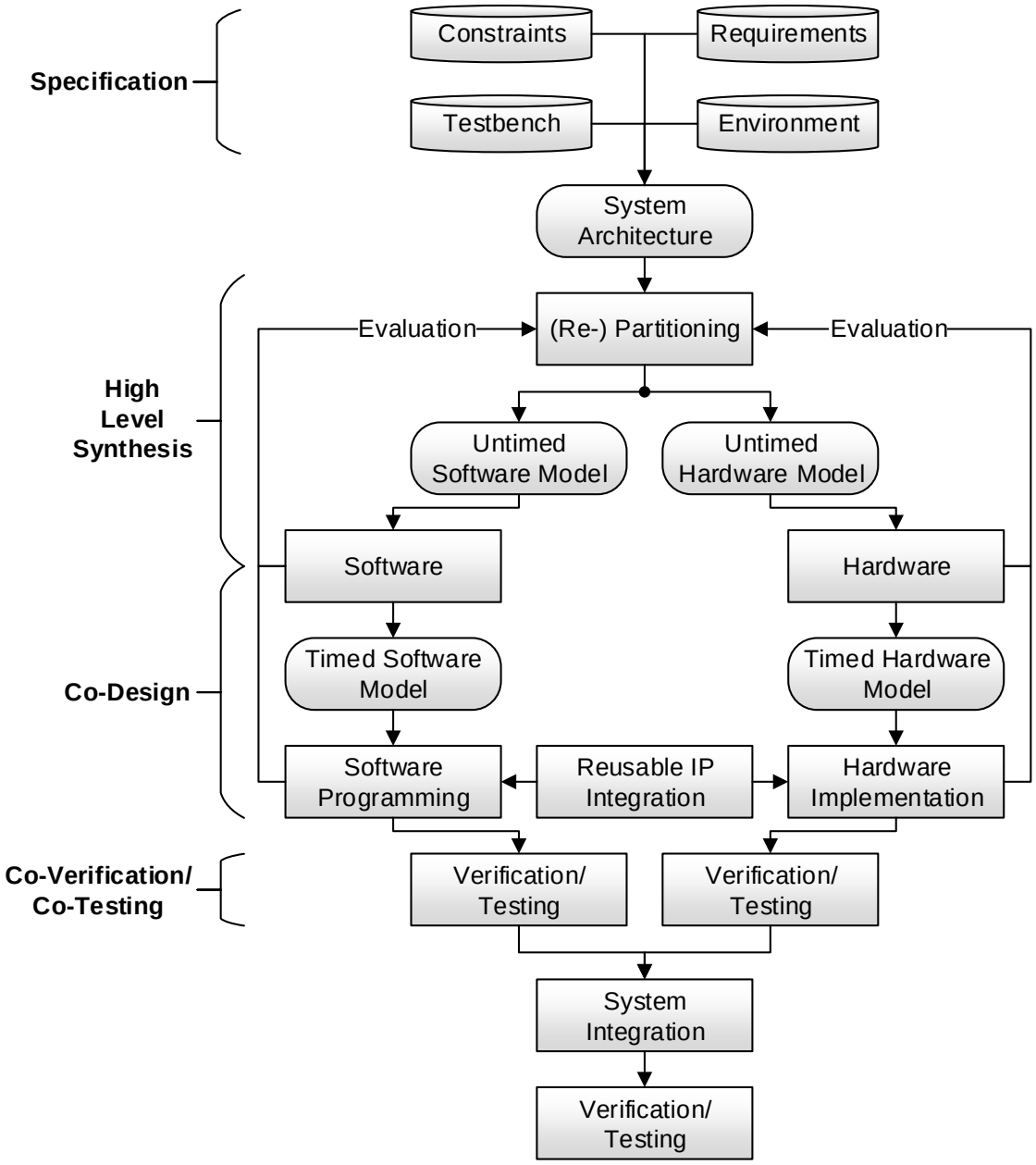



Abbildung 3.7: Hardware-Software Co-Design

der Entwicklung gibt es dann verschiedene gezeitete Modelle. Hierbei wird nach [CLHH10] unterschieden:

- Untimed: keine explizites Timing, aber es wird das gleichzeitige und sequentielle Ablaufen von Operationen beachte.
- Loosely Timed: beinhaltet nur minimale zeitliche Informationen um beispielsweise multiple Threads ohne explizite Synchronisierung zu verwalten.
- Approximately Timed: es existiert ein direktes Mapping von extern beobachtbaren Zuständen und der dazugehörigen Repräsentation im Modell.

- Cycle Accurate: der zukünftige Zustand des Modells kann zu jedem beliebigen Zeitpunkt bestimmt werden und es existiert eine eins zu eins Korrespondenz zwischen den Zuständen des Modells und dem zugehörigen RTL-Modell (*Register Transfer Level* (*RTL*)) zu jedem Takt.

Ein wichtiger Aspekt bei der parallelen Entwicklung von Hard- und Software ist die Fehlersuche. Aufgrund der steigenden Komplexität von beiden Teilen steigt auch die Komplexität von benötigten Testumgebungen. Aus diesem Grund müssen neue Möglichkeiten untersucht werden, wie man die Fehler in den jeweiligen Designs finden und beheben kann.

## 3.5 Fazit - Softcore Prozessoren für echtzeitkritische Daten- und Bildverarbeitung

Die Leistungsfähigkeit von eingebetteten Systemen steigt rasant an und die Steigerung entspricht dem Moore'schen Gesetz. Aus diesem Grund können komplexe Aufgaben, die bisher nicht durch eingebettete Systeme lösbar waren, mit diesen berechnet werden. Viele dieser Aufgaben fallen in den Bereich der echtzeitkritschen Datenverarbeitung und im Speziellen der echtzeitkritischen Bildverarbeitung. Mit der steigenden Komplexität müssen neue Verfahren, Methoden und Modelle entwickelt werden, die sich dieser Problematik und vor allem den neuen Problemfeldern anpassen können. Eine kostengünstige Alternative sind die hier vorgestellten SoRCs, als Teilmenge der SoCs, die Aufgrund ihrer Rekonfigurierbarkeit wesentliche Vorteile bei Entwicklungszeiten und damit auch Entwicklungskosten bieten.
In Kapitel 3 wurden die wesentlichen Grundlagen für die allgemeine Entwicklung von Softcore Prozessoren erörtert. Es wurden wesentliche Merkmale des Gesamtentwurfs diskutiert und Entwurfswege von IP Logik für Softcores vorgestellt sowie die allgemeine Synthese erläutert.
Der wesentliche Kostenfaktor bei jeder Entwicklung eines eingebetteten Systems ist die benötigte Entwicklungszeit, die TTM. Mit Hilfe der vorgestellten SoRCs lässt sich diese bereits stark reduzieren. Wenn innerhalb eines SoRCs mindestens ein Softcore zum Einsatz kommt, ist es aufgrund der Anpassbarkeit dieser Prozessoren möglich, den Wiederverwendungsfaktor weiter zu steigern und damit die Entwicklungszeit weiter zu verkürzen. Ein Hardware-Software Co-Design wie in 3.4 vorgestellt kann an dieser Stelle Verwendung finden.
Wenn diese SoRCs mit Softcore Bestandteilen in den unter 2.6 vorgestellten Problemfeldern eingesetzt werden sollen, kommt es aufgrund der Anforderungen dieser Algorithmenklassen zu einer Reihe von Problemen. Allgemeine Softcore Prozessoren, wie beispielsweise der Leon3, besitzen aufgrund ihrer universellen Einsetzbarkeit wesentliche Defizite, die eine Verwendung in den vorgestellten Algorithmenklassen ausschließen. Die Prozessoren selbst verfügen nicht über die benötigte Genauigkeit, entsprechende Prozessoren arbeiten nur mit der Single-Genauigkeit (32 Bit). Zusätzlich ist die komplette taktgenaue Vorhersagbarkeit und ein komplett planbares Verhalten nicht Kerncharakteristik vieler dieser Softcores. Für die schnelle und damit kostengünstige Lösung solcher Probleme wird eine komplette Verarbeitungskette zur Erstellung von Softcores benötigt, die mit Hilfe einer speziell angepassten Methodik einen Entwurfsweg von hoch abstrahierten Modellen bis zur Logik-Ebene realisiert. Hierbei müssen zur Maximierung der Wiederverwendbarkeit sämtliche Verarbeitungseinheiten des Softcore modular austauschbar sein. Um die maximale Verarbeitungsgeschwindigkeit erreichen zu können, ist es notwendig, vergleichsweise viel Ressourcenfläche jeder Verarbeitungseinheit zur Verfügung stellen zu können. Um dennoch die notwendige Breite an verwendbaren

Verarbeitungseinheiten zu erzielen, ist ein modularer und partieller Austausch sinnvoll. Dieser muss über ein Modell zur Designzeit zeitlich planbar sein. Aufgrund der algorithmischen Eigenschaften sind speziell die Bereiche der Bildverarbeitung sehr gut zur Rekonfiguration geeignet, da sich die Verarbeitungsalgorithmik über die Zeit dominant ändert und somit eine zeitliche Anpassung sinnvoll ist, gerade um die Rechenzeiten innerhalb einer Konfiguration minimieren zu können.
Um das zeitliche Verhalten vorhersagen zu können, ist die Entwicklung einer speziell angepassten Assemblersprache sowie eine toolgestützte Erstellungs- und Verarbeitungskette dieser notwendig. Hierbei sollte es möglich sein, komplette Modelle in Maschinencode überführen zu können und dabei den entstehenden Abstraktionsoverhead zu minimieren.
Die Verarbeitungsgeschwindigkeit des Softcores ist dabei maßgeblich von dem im Assembler erzielten Parallelisierungsgrad der Operationen abhängig. Da die Optimierung von komplexen Algorithmen dieser Größenordnung einen weiteren aktuellen Forschungsschwerpunkt darstellt, ist es notwendig sinnvolle Mechaniken zur Maximierung der Verarbeitungsgeschwindigkeiten in die Übersetzungsmodelle einzuarbeiten.

# 4 Softcore-IPs im Rahmen eines modellbasierten Entwurfs für System-on-Reprogrammable-Chips

Wird es innerhalb eines Designprozesses von SoRCs notwendig ein Re-Design durchzuführen, kann dies sehr zeit- und kostenintensiv werden. Hierbei gibt es zwei Problempunkte, wenn von der ursprünglichen Spezifikation abgewichen wird, bei Änderungen bei der geplanten Funktionalität und bei den spezifizierten notwendigen Leistungsanforderungen. Mit der Hilfe des in Abschnitt 3.1 unter Abbildung 3.2 vorgestellten Y-Diagramms für die Entwicklung auf FPGAs kann das Problem der sich ändernden Leistungsanforderungen durch die Analyse der Systemperformanz und der verwendeten Architektur effektiv gelöst werden.
Kommt es zu einer notwendigen Änderung bei der geplanten Funktionalität, so bietet das in Abschnitt 2.1 unter Abbildung 2.2 vorgestellte V-Modell die Möglichkeit die größtmögliche Schnittmenge beider Spezifikationen herauszufinden und damit nur in einzelnen Partitionen gezielte Änderungen vornehmen zu müssen. Allerdings ist das allgemeine V-Modell nicht für die Verwendung in SoRC-Systemen spezifiziert und es müssen daher einige im folgenden erläuterte Anpassungen vorgenommen werden, um eine bessere Flexibilität bei der Entwicklung zu gewährleisten.

Allerdings ist die Erstellung eines Designs auf dem FPGA nur ein Teil bei der Entwicklung von SoRCs. Zum einen muss nicht nur eine Hardwarearchitektur auf dem FPGA erzeugt werden, sondern sowohl eine dazu gehörende Software, als auch die dem FPGA umgebende Logik muss entwickelt werden. Hierfür eignet sich das V-Modell mit einigen im Folgenden vorgestellten Anpassungen.

## 4.1 Das Phi-Modell zur Softcoreentwicklung

Bei der Entwicklung von Softcore Prozessoren wird ein alternatives Modell vorgeschlagen. Das Phi-Modell, in Anlehnung der Form des Modells an den griechischen Buchstaben Φ, im Folgenden als Φ-Modell bezeichnet, wurde entworfen um die speziellen Anforderungen bei der Softcore Prozessorentwicklung abzubilden. Das Modell ist als Teilmodell bzw. Untermodell eines angepassten V-Modells zur SoRC Entwicklung zu verstehen und wird in Abschnitt 4.2 in das entsprechend modifizierte V-Modell eingeordnet.
Das Φ-Modell ist das Ergebnis der Analyse eines allgemeinen in Abschnitt 3.4 unter Abbildung 3.7 vorgestellten Hardware-Software Co-Design Ansatzes. Die Entwicklung von Softcore Prozessoren ist eine Kombination aus Hardware und paralleler Softcore-Software Entwicklung. Allerdings gibt es bei der Entwicklung von Softcore Prozessoren eine stärkere gegenseitige Beeinflussung von Hardware und Software, weswegen ein allgemeines Modell wie in Abbildung 3.7 nicht geeignet ist.

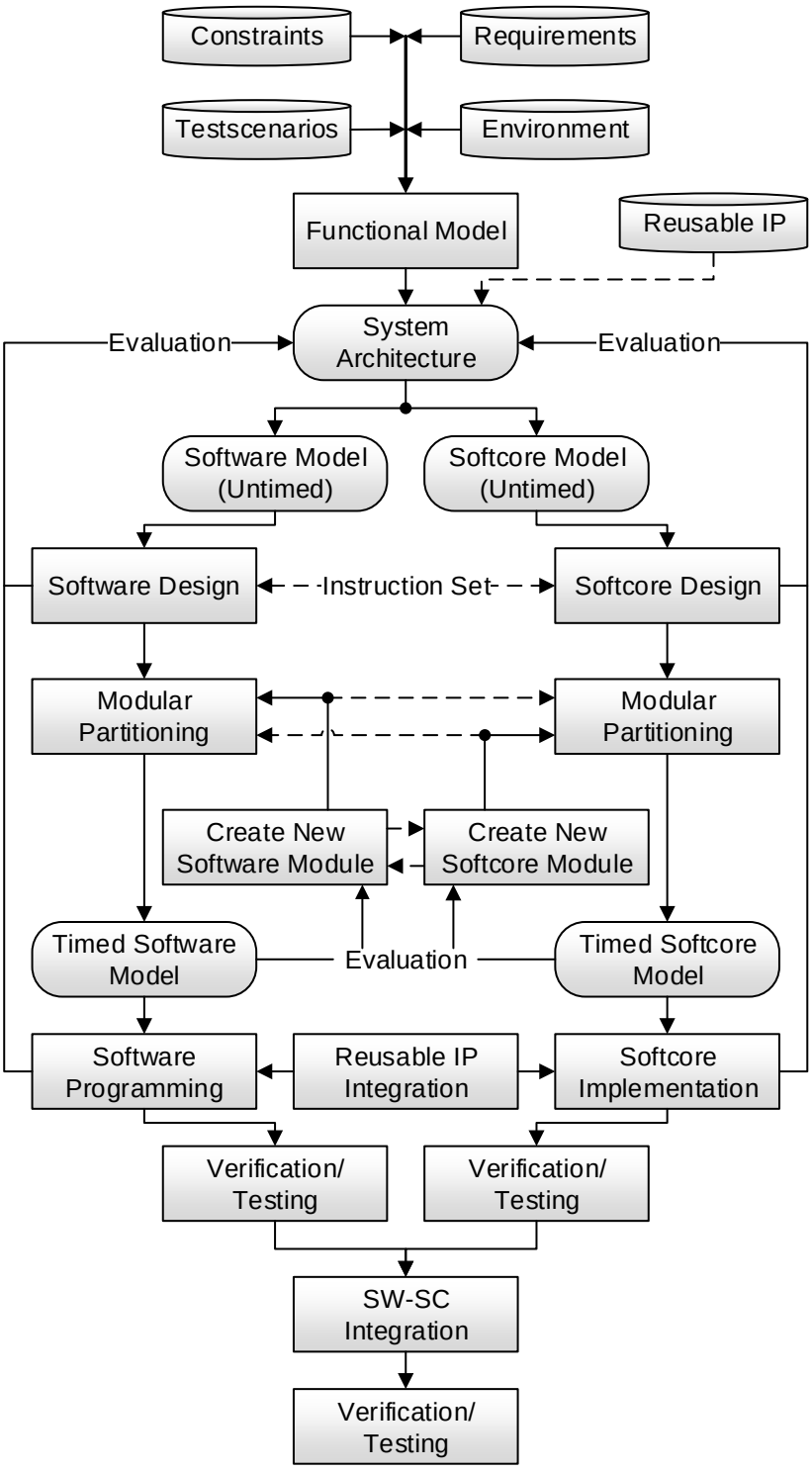



Abbildung 4.1: Φ-Modell für die Softcoreentwicklung

Bei dem Φ-Modell der Softcoreentwicklung wird ein Funktionsmodell des Softcore (*Functional Model*) aufgrund der aufgabenspezifischen Anforderungen (*Requirements*), etwaiger Einschränkungen (*Constraints*) sowie zuvor entwickelter Testszenarien (*Testscenarios*) und der umgebenden Logik (*Environment*) ausgewählt und wird als (Teil-) Spezifikationssatz als Eingangsschnittstelle von dem übergeordneten V-Modell bereitgestellt. Aus diesem Funktionsmodell (*Functional Model*) wird unter Einbeziehung etwaiger bereits entwickelter oder durch Dritte bereitgestellter IPs (*Reusable IP*) eine (vorläufige) Systemarchitektur (*System Architecture*) modelliert. Die Systemarchitektur wird im Anschluss innerhalb dieses Modells verfeinert und es erfolgt eine Parallelentwicklung der Hardware, also dem eigentlichen Softcore, und einen Softcoresoftwareteil. Hierbei ist im weiteren Verlauf des Abschnitts die Softcoresoftware mit Software abgekürzt, um die Lesbarkeit zu verbessern. Im Folgenden wird ein Hardware-Software Co-Design Ansatz verfolgt, wie bereits allgemein in Abschnitt 3.4 vorgestellt. Dieser wird allerdings aufgrund der speziellen Eigenschaften von Softcore Prozessoren modifiziert, wie im Weiteren im Detail erläutert wird. Eine Modifikation ist sinnvoll, da aufgrund der speziellen Eigenschaften von Softcore Prozessoren viele Lösungsvariationen möglich sind und damit kürzere Evaluationszyklen innerhalb der Entwicklung zu einer effizienteren Entwicklung führen. Diese Parallelentwicklung ist bei Softcore Prozessoren sinnvoll, da diese sowohl aus einem Hardwareteil, also dem eigentlichen Softcore, als auch aus einem Softwareteil samt spezifischen Befehlssatz (*Instructuin Set*), bestehen. Der Funktionsumfang und dem gegenüberstehend der Befehlssatz samt realisierter Software bedingen sich gegenseitig und sind abhängig von den jeweiligen Anforderungen. Diese gegenseitige Abhängigkeit und die damit verbundene parallele Entwicklung sowohl der Software- als auch der Hardwarekomponenten machen ein softcorespezifisches Entwicklungsmodell sinnvoll, um die Effizienz bei der Entwicklung zu verbessern.
So ist es möglich, neue Softcoremodule zur Erweiterung und Anpassung der ALU sehr effizient umsetzen zu können, was eine potenzielle Anpassung der parallel entwickelten Software mit sich bringt und zu einer Anpassung des Systems führen kann. Ein anderes Szenario ist die softwareseitige Feststellung des frequentierten Verwendens einer speziellen Abfolge von Befehlen und die Möglichkeit der Umsetzung dieser Befehlsfolge in einem angepassten separierten Hardwareoperator innerhalb der ALU. Da sich die Hard- und Softwarekomponenten hier sehr stark gegenseitig beeinflussen und es möglich ist beide Komponentenseiten aufeinander anzupassen, wurde ein zusätzlicher verkürzter Evaluationsschritt eingeführt und die Entwicklung neuer Module als wichtiger zu separierender Punkt aufgefasst und entsprechend im Modell dargestellt.
Nach der anfänglichen Partitionierung der Softcore- und Softwarekomponenten erfolgt die Erstellung eines Softcorefunktionsmodells, basierend auf der festgelegten Systemarchitektur (*Softcore Model (Untimed)*), welches kein exaktes Zeitverhalten besitzt und nur die korrekte Aufteilung der Funktionalität verifizieren soll. Basierend auf diesem Modell werden die einzelnen Module der Softcorearchitektur (*Modular Partitioning*) abgeleitet. Mit Hilfe der Moduldefinitionen und der festgelegten vorläufigen Funktionsweise jedes Moduls ist es möglich, ein gezeitetes Softcoremodell (*Timed Softcore Model*) zu erstellen und dieses mit den Spezifikationen (*Requirements*) abzugleichen. Um mit Hilfe dieses Modells ein aussagekräftiges Ergebnis zu erhalten, muss innerhalb des Modells ein direktes Mapping von extern beobachtbaren Zuständen und der dazugehörigen Repräsentation im Modell existieren.
Parallel zu dem Softcoreentwicklungszweig findet der Softwareentwicklungszweig für die Softcoresoftware mit vergleichbaren Schritten statt. Direkt nach der Partitionierung in Softcore- und Softwarekomponenten wird ein Funktionsmodell (*Untimes Software Model*) erstellt, das die geplante Software in einzelne logisch/funktionell zusammengehörige Teile gliedert, die im

Weiteren als Softwaremodule bezeichnet werden (*Modular Partitioning*). Zu diesem Zeitpunkt kann ein Befehlssatz (*Instruction Set*) festgelegt werden, dass das Softcore Design beeinflusst bzw. genauere Vorgaben über den notwendigen Funktionsumfang des Prozessors liefert.
Unter Verwendung der Softwaremodule wird ein detaillierteres Softwaremodell erstellt, welches möglichst genaue Zeitvorgaben für jedes einzelne Modul beinhaltet. Mit Hilfe der zeitlichen und funktionellen Vorgaben (*Constraints*) kann nun jedes Modul auf seine Einhaltung dieser Parameter hin überprüft werden. Es erfolgt parallel mit dem aktuellen Entwicklungsstand der Softcorehardware eine Evaluation (*Evaluation*). An dieser Stelle, im Unterschied zum originalen Co-Design Ansatz, findet eine Evaluation zwischen den jeweiligen Modulen statt, um etwaige Problempunkte des einen Entwicklungszweigs mit Hilfe der Anpassung der Module des anderen Entwicklungszweigs lösen zu können. Denkbar wären beispielsweise neue Hardwaremodule, welche die zeitliche Verarbeitung einzelner Softwaremodule beschleunigen, um etwaige zeitliche Einschränkungen nicht zu verletzen, oder die Verteilung eines Softwarebefehls auf funktionsgleiche andere Befehlsfolgen, um Hardwareoperatoren und damit Chipfläche einsparen zu können. Weiterhin ist auch eine einfache Anpassung der verwendeten Chipfläche der einzelnen Softcoremodule möglich, je nach Frequenz der dazugehörigen Softwarebefehle, um eine zeitliche oder chipflächen Optimierung zu erzielen. Auch die Anpassung einzelner Software- oder Softcoremodule ohne die Notwendigkeit die Gegenstücke zu verändern, ist denkbar, beispielsweise um die Wiederverwendung bereits realisierter IP-Blöcke zu ermöglichen.
Dieser zusätzliche Evaluationsschritt der gezeiteten Modelle (*Timed Software Model* und *Timed Softcore Model*) und die daraus folgenden etwaig notwendigen neu konzipierten Module sowohl auf der Softwareseite (*Create New Software Module*), als auch auf der Softcoreseite (*Create New Softcore Module*) ermöglichen eine Senkung des Zeitbedarfs im Projekt, da diese Evaluierung und Anpassung des Konzepts vor der eigentlichen Implementierung bzw. Programmierung ausgeführt werden. Eine Anpassung des Systems vor diesen Schritten verhindert eine zeitaufwendige Anpassung bereits realisierter Komponenten. Da sich viele solcher Komponenten gegenseitig beeinflussen und bedingen, wird so die Anpassung vieler einzelner Module verhindert.
Erfüllen die jeweiligen gezeiteten Modelle (also das *Timed Software Model* und das *Timed Softcore Model*) alle Anforderungen, so kann eine Implementierung des Softcores (*Softcore Implementation*), bzw. eine Programmierung der Software (*Software Programming*) vorgenommen werden. An der Stelle der praktischen Realisierung fließen bereits erstellte IP-Blöcke in die Entwicklung ein (*Reusable IP Integration*), um den Realisierungsaufwand und die Realisierungszeit minimieren zu können. Je nach Projekt können von einzelnen softwaresetigen Befehlsfolgen bis hin zu kompletten Softcore Prozessoren entsprechende Module wiederverwendet werden. Im Anschluss an die Programmierung der Software erfolgt ein weiterer Evaluationsschritt (*Evaluation*) mit Hilfe einer eins zu eins Korrespondenz zwischen den Zuständen des Programms und dem zugehörigen RTL-Modell zu jedem Takt. Etwaige Verletzungen der Spezifikation werden erkannt und in einem weiteren Durchlauf durch das Φ-Modell ab der Erstellung der Systemarchitektur (*System Architecture*) behoben.
Bevor die realisierten Softwarekomponenten in den umgesetzten Softcore integriert wird (*SW-SC Integration*), erfolgt eine separierte Verifikation bzw. Ausführung von zuvor definierten Testfällen (*Verification/Testing*) zur Feststellung der korrekten Funktionalität und des taktgenauen Verhaltens. Im Anschluss an die Integration der Software in den Softcore Prozessor erfolgt ein weiterer Verifikationsschritt (*Verification/Testing*), um die korrekte Funktionsweise des gesamten Softcore mit der speziellen für das Projekt entwickelten Software nachzuweisen.

## 4.2 Das angepasste V-Modell zur SoRC Entwicklung

Die Erstellung und (Wieder-) Verwendung eines Softcore Prozessors in einem Gesamtsystem ist nur ein Teil bei der Entwicklung von SoRCs. Üblicherweise löst ein Softcore eine spezielle Teilaufgabe in einem entsprechenden System und ist von weiterer FPGA Logik umgeben, die wiederum von weiterer Logik umgeben sein kann. Ein Softcore ist damit als Teilsystem in einen FPGA eingebettet, dieser wiederum kann als Teilsystem in einem Gesamtsystem eingebettet sein. Aufgrund der sich dadurch ergebenden Hierarchie, ist es in modernen komplexen System notwendig, eine hierarchisch und modulare Entwicklung, mit definierten Eingabe- und Ausgabeverhalten in jeder Hierarchieebene, zu ermöglichen. Die verwendeten Modelle zur Realisierung von Projekten dieser Art müssen sich deshalb an die komplexeren Aufgabenstellungen anpassen und mit diesen skalieren können. Je nach (Teil-) Aufgabe sind verschiedene Modelle besser geeignet, wie in den vorhergehenden Kapiteln der Arbeit vorgestellt. Deswegen wird im Weiteren eine hierarchische Kombination verschiedener angepasster Modelle vorgeschlagen, um für jede Teilaufgabe eine möglichst effiziente Entwicklung zu garantieren und die Möglichkeit zu schaffen, ein skalierbares Gesamtmodell zu erhalten. Ein wichtiges Augenmerk liegt dabei auf modularen Einheiten, also einzelnen Teilmodulen mit definierten Eingabe- und Ausgabeverhalten, die jeweils ausgetauscht werden können. Entsprechende Änderungen dieser Module sollten dabei keinen oder nur einen geringen Effekt auf andere Module haben. Dieses Black-Box ähnliche Verhalten sichert ein effizientes Austauschen von einzelner Logik, ohne der Notwendigkeit das gesamte System bzw. das jeweils übergeordnete System anpassen und erneut testen zu müssen. Werden die auf der selben Ebene befindlichen Tests erfolgreich ausgeführt, so kann davon ausgegangen werden, dass sich das Systemverhalten insgesamt funktional genauso verhält, und bereits erfolgreich abgeschlossene Tests aus übergeordneten Ebenen nicht wiederholt werden müssen.
Die Idee des folgenden Modells ist eine Maximierung der Skalierbarkeit und damit möglichst effiziente Entwicklung mittels eines „Divide and Conquer“-Ansatzes innerhalb des Partitionierungsschrittes.

Wie in Abbildung 4.2 zu sehen ist, besteht der Einstieg einer SoRC-Systementwicklung aus einem allgemeinen V-Modell. Die Definition der Anforderungen, Einschränkungen und Spezifikationen (*Constraints/Requirements Analysis*) des Gesamtsystems ist der erste und wichtigste Schritt. Hieraus leiten sich direkt die Testszenarios (*Testscenarios*) für den abschließenden Annahme- bzw. Übergabetest (*Acceptance Test*) ab. Das funktionelle Design (*Functional Design*) sowie der sich auf der selben Ebene befindliche Funktionstest (*Functional Test*) sind ebenfalls unverändert. Die Partitionierung (*Partitioning*) erfolgt in diesem Modell allerdings in zwei nacheinander ablaufenden Schritten und teilt das Modell in weitere funktionelle Ablaufsmodelle:

- Im ersten Schritt wird das funktionelle Design in ein (optionales) umgebendes nicht-FPGA Design und ein Design, welches auf einem FPGA umgesetzt werden soll, unterteilt. An dieser Stelle ist das Teildesign des FPGA als Black-Box Komponente zu betrachten, welches festgelegte Ein- und Ausgänge verwendet und ein spezifiziertes Systemverhalten besitzt. Mit Hilfe dieser Black-Box Definition kann das dem FPGA umgebende Design, das einbettende System, entwickelt werden. Hierzu soll aus hierarchischen Gründen, und je nach Komplexität, ein weiteres V-Modell, wie in Abschnitt 2.1 unter Abbildung 2.2 vorgestellt, verwendet werden. Ein wesentlicher Punkt ist die saubere Trennung der zu Beginn festgelegten Anforderungen und Einschränkungen zu einem für den FPGA-Teil geltenden Spezifikationssatz und den Spezifikationen, die für die umgebende Logik gelten

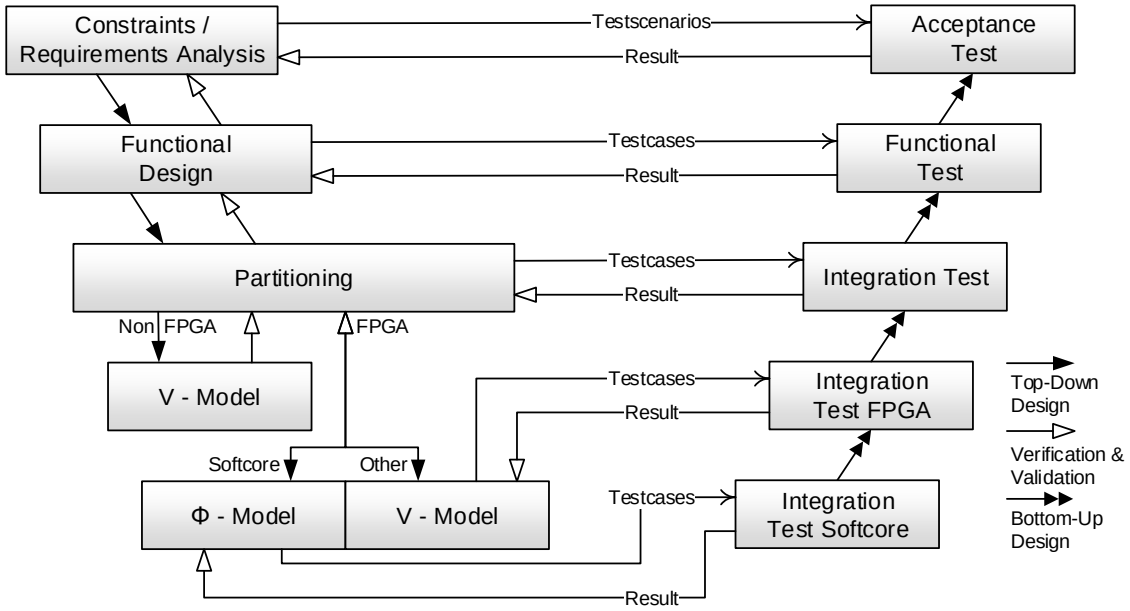



Abbildung 4.2: V-Modell für eingebettete Systementwicklung mit Softcore Prozessoren

müssen. Für die umgebende Logik des Gesamtsystems können, je nach Anforderungsprofil, verschiedene Vereinfachungen getroffen werden. An dieser Stelle sollten die Spezifikationen ebenfalls präzisiert werden.

- Im zweiten Schritt der Partitionierung (*Partitioning*) wird die geplante FPGA-Funktionalität unterteilt in einen Softcore, der eine definierte Aufgabe lösen soll, und eine dem Softcore umgebende weitere FPGA-Logik. Der Softcore selbst wird ebenfalls auf dieser Ebene als Black-Box Modul angesehen und mittels eines in Abbildung 4.1 vorgestellten Φ-Modells entwickelt. Die dem Softcore umgebende Logik wird an dieser Stelle, je nach Komplexität, mittels des vorgestellten V-Modells entwickelt. Da der Softcore selbst zwar ein Teil dieser Logik ist, aber als Black-Box betrachtet wird, kann eine entsprechende Entwicklung vorgenommen werden ohne den final verwendeten Softcore fertig entwickelt zu haben. Weiterhin können Softcore unabhängige Tests innerhalb dieses Unter-V-Modells durchgeführt werden, da der Softcore ein spezifiziertes Ein- und Ausgabeverhalten hat, das simuliert werden kann. Je nach Komplexität sollte eine weitere Präzisierung und eine etwaige Aufteilung der Spezifikationen erfolgen.

Die Definition von präzisierten Anforderungen, Einschränkungen und Spezifikationen (*Constraints*, *Requirements*), sowie deren (optionale) Aufteilung auf die einzelnen Komponenten ist dabei der Einstiegspunkt in die untergeordneten V-Modelle und der finale Test (*Acceptance Test*) der jeweiligen Unter-V-Modelle von Nicht-FPGA Logik und Nicht-Softcore FPGA Logik fungiert als Übergabeschnittstelle der jeweiligen Modelle in das übergeordnete V-Modell. Diese Tests befinden sich auf unterschiedlichen Ebenen auf der rechten Seite des V-Modells und werden Bottom-Up in folgender Reihenfolge ausgeführt: der Integrationstest der einzelnen Softcore Komponenten zu einem funktionierenden Softcore Prozessor (*Integration Test Softcore*), die Tests zur Einbettung des Softcore in die umgebende FPGA Logik (*Integration Test FPGA*) und die Integration des gesamten FPGA in eine etwaige Gesamtsystemlogik (*Integration Test*).
Den Eintritt in die Softcore Entwicklung und damit den Eintritt in das Φ-Modell bilden

ebenfalls die Definition von präzisierten Anforderungen, Einschränkungen und Spezifikationen (*Constraints*, *Requirements*), sowie die Definition der dem Softcore umgebenden FPGA-Logik (*Environment*) und die Testszenarien (*Testscenarios*), mit Hilfe derer der Softcore getestet werden soll. Der mittels des Φ-Modells entwickelte und getestete Softcore Prozessor wird nach einem finalen Verifikationsschritt (*Verification/Testing*) an das übergeordnete V-Modell übergeben. Dieser Test bildet damit die Übergabeschnittstelle und ist in Abbildung 4.2 als Integrationstest des Softcore (*Integration Test Softcore*) dargestellt.

## 4.3 Methodik einer Softcore-Toolchain

Da es sich bei der unter Abschnitt 2.6 vorgestellten Problemklasse der entwickelten Softcore Prozessoren um eine Klasse handelt, deren Spezifikationen sich innerhalb eines definierten Spezifikationsrahmens bewegen, ist es möglich große Teile vorangegangener Projekte wiederzuverwenden, sofern diese bei der Realisierung entsprechend konzipiert wurden. In den meisten Fällen sollte es sogar möglich sein, einen erheblichen Teil eines zuvor entwickelten Softcore Prozessors wiederzuverwenden. Aus diesem Grund, und um die Wiederverwendbarkeit zu maximieren, ist es sinnvoll eine methodische Verarbeitungskette (*Toolchain*) zu entwickeln, die entsprechend des Spezifikationsrahmens der adressierten Problemklassen möglichst generische Eigenschaften besitzt, und trotzdem im Hinblick der eingesetzten Problemklasse spezialisiert ist. Vorgeschlagen wird aus diesem Grund eine Softcore-Bibliothek (*Softcore Library*) mit mindestens einem auf die Aufgabenklasse angepassten und spezialisierten Softcore, der in verschiedenen generischen Variationen vorliegt und innerhalb der Problemklasse der Echtzeitdatenverarbeitung bzw. der echtzeitkritischen Bildverarbeitung auf jedes Projekt hin angepasst werden kann. Die generische Anpassbarkeit innerhalb der spezifizierten Aufgabenklasse sorgt für ein spezielles Design, das eine Wiederverwendbarkeit und eine Anpassbarkeit maximiert, ohne die grundlegend geforderten Eigenschaften, wie beispielsweise die Forderung nach harter Echtzeitfähigkeit, zu verletzen. In diesem Zusammenhang ist es möglich, nicht nur eine Softcore-Bibliothek zu erstellen und mit jedem neuen Projekt auszubauen, sondern es ist auch sinnvoll eine ganze Verarbeitungskette zu entwickeln. Diese kann in entsprechenden Projekten Einsatz finden und es ist möglich, auf die Ergebnisse vorangegangener Projekte effizient zuzugreifen. Entsprechend wird an dieser Stelle davon ausgegangen, dass bereits ein Softcore Prozessor nach dem vorgestellten Φ-Modell aus Abbildung 4.1 entwickelt wurde und das wesentliche Teile eines Softcores bereits in einem IP Katalog (*IP Catalog*) zusammengetragen wurden.

Diese Verarbeitungskette muss in verschiedenen Abstraktionsebenen realisiert sein und sollte bei jedem Zwischenschritt möglichst breit verwendete Schnittstellen zu anderen Programmen bieten. Der Ansatz erfolgt auf der höchsten hier verwendeten Abstraktionsebene: Nach der projektspezifischen Aufgabendefinition (*Task*) wird unter Zuhilfenahme der definierten Anforderungen und Einschränkungen (*Requirements*, *Constraints*) ein Modell des zu entwickelnden Algorithmus entworfen. Dieser Entwurf erfolgt auf einer modellbasierten Abstraktionsebene und wird mittels eines Datenflussgraphen (*Dataflow Model*) realisiert. Die Modellierung des Datenflussgraphen sollte mit einem in der Industrie weit verbreiteten Modellierungstool erfolgen (*Dataflow Model Generation*), exemplarisch sind hier Matlab/Simulink oder LabVIEW als wichtigste Vertreter aufzuführen. Denkbar wären aber auch andere in der Forschung und Entwicklung eingesetzte

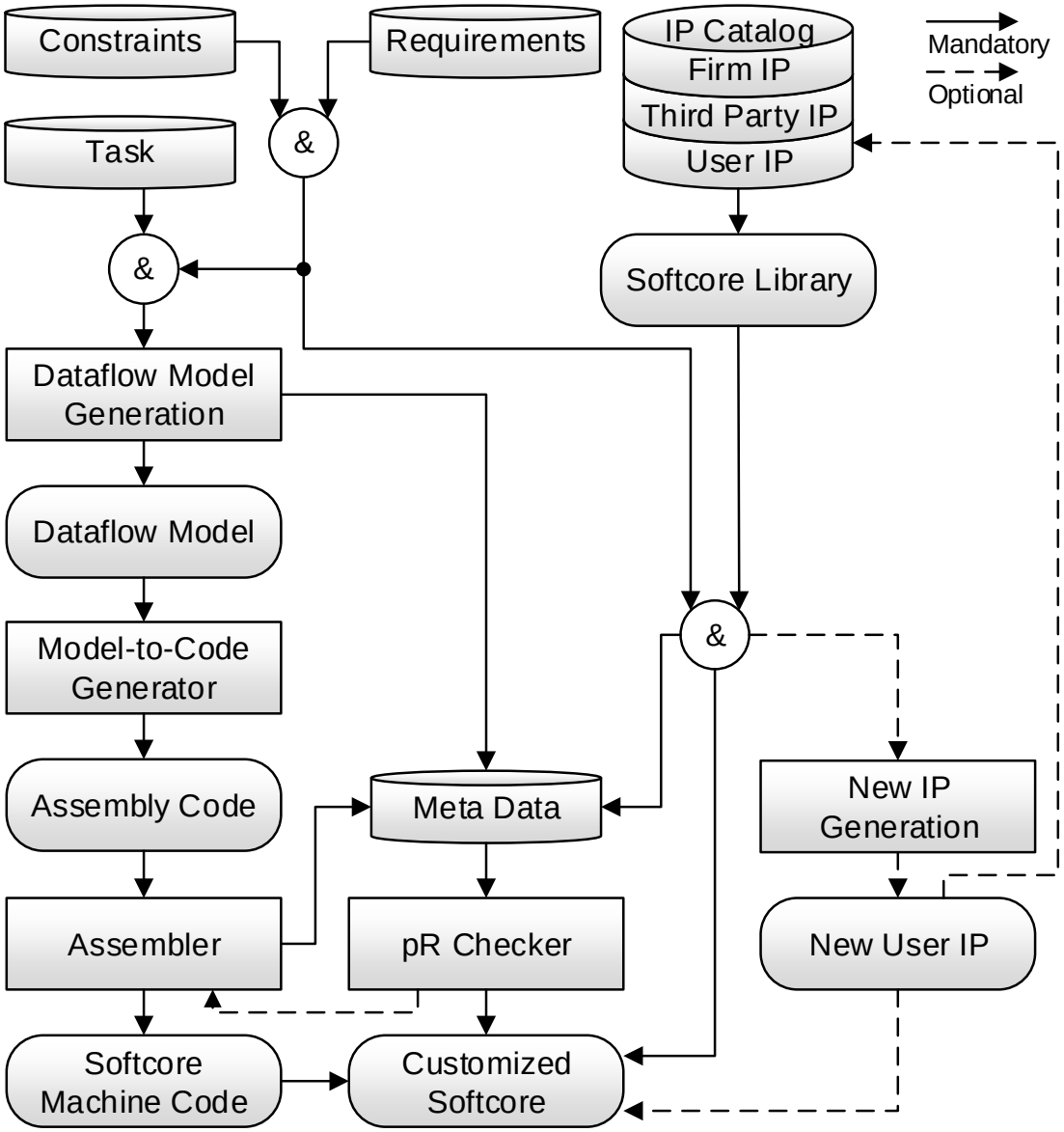



Abbildung 4.3: Toolchain zur Softcore Anpassung und Softcore-Softwareentwicklung

Tools wie beispielsweise Scilab/Xcos.
Aus diesem Modell wird dann über den Assemblercodegenerator (*Model-to-Code Generator*) der Assemblercode (*Assembly Code*) aus dem vorliegenden Modell generiert. Sowohl bei dem Schritt der Modellgenerierung (*Dataflow Model Generation*), als auch beim Schritt der Übersetzung von Modell zu Assemblercode (*Model-to-Code Generator*) finden verschiedene Optimierungen statt, auf die genauer in Abschnitt 4.5 eingegangen wird.
Der Assemblercode wird im Anschluss von einem speziell für den in der Aufgabenklasse verwendeten Befehlssatz-angepassten optimierenden Assembler (*Assembler*) in Softcore-lesbaren Maschinencode übersetzt. Der Maschinencode bildet die niedrigste hier verwendete Abstraktionsebene und wird direkt vom Softcore Prozessor verwendet. Verschiedene Methodiken zu Schedulingstrategien und anderen Optimierungsansätzen werden dabei in Abschnitt 4.6

vorgestellt.
Eine Unterscheidung in Assemblercode und Maschinencode ist sinnvoll, da der Assemblercode lediglich eine sequentielle Abfolge des zugrunde liegenden Datenflussgraphen ist, der Maschinencode allerdings verschiedene Mechaniken verwendet, die eine gepipelinte Verarbeitung realisieren und in diesem Zusammenhang das Aufteilen der sequentiellen Assemblercodebefehle in Operatorstartbefehle und Ergebnisspeicherbefehle.
Die Aufteilung in ein Modell bzw. einen Graphen, der entsprechende Parallelitäten abbilden kann, einen sequenziellen Assemblercode und einen Maschinencode ist deshalb notwendig, da in jeder Abstraktionsebene verschiedene Informationen vorhanden sind und verwendet werden, aber auch verschiedene Informationen absichtlich nicht vorhanden sind. So verwendet der Graph keine Informationen darüber, wie die Realisierungsdetails der jeweiligen Operationen aussehen, ob diese parallelisierbar sind oder andere spezielle Eigenschaften besitzen. Diese Informationen sind aber für optimierende Schedulingstrategien notwendig. Aus diesem Grund existieren auf jeder Abstraktionsebene verschiedene Optimierungen, die mit den jeweilig auf diesen Ebenen vorhandenen Informationen arbeiten.
Wie bereits erläutert existiert eine Softcore-Bibliothek (*Softcore Library*) mit verschiedenen einsetzbaren Modulen (*IP Catalog*). Diese Module können von externen Entwicklern (*Third Party IP*) entworfen worden sein, bereits als Entwicklungen aus vorangegangenen Projekten vorliegen (*User IP*) oder innerhalb des aktuellen Projektes neu entwickelt werden (*Create New IP*).
Zur Anpassung der Softcore-Bibliothek (*Softcore Library*) zu einem auf das aktuelle Projekt hin optimierten Softcore (*Customized Softcore*) werden entsprechende Meta-Informationen (*Meta Data*) benötigt. Diese ergeben sich zum einen aus den definierten Anforderungen und Einschränkungen (*Requirements*, *Constraints*) und aus den Rahmenbedingungen der zur Verfügung stehenden Softcores, aber auch aus den Informationen, die sich bei der Erstellung des Datenflussgraphen (*Dataflow Model Generation*) bzw. dem vom Assembler (*Assembler*) übersetzten Ergebnis des Assemblercodes, dem Maschinencode (*Softcore Machine Code*), ergeben. Diese Informationen können von Projekt zu Projekt variieren, aber wesentliche Kerninformationen sind die definierten Einschränkungen, beispielsweise die zu erreichenden Echtzeitschranken, maximal verfügbare Ressourcen oder die aktuell verfügbaren bzw. verwendeten Operationen. Diese Operationen beinhalten dann weitere Informationen wie benötigte Ressourcen und Taktzyklen jedes Operators. Zu diesen Metainformationen zählen aber auch Informationen über das zeitabhängige Anforderungsprofil des Algorithmus.
Mit Hilfe dieser Informationen kann ein partieller Rekonfigurationschecker (*pR Checker*) berechnen, ob eine Softcorerekonfiguration für das aktuelle Projekt sinnvoll ist und an welchen Stellen ggf. rekonfiguriert werden kann. Je nach Ergebnis kann es notwendig sein, dass der vom Assembler generierte Maschinencode (*Softcore Machine Code*) angepasst werden muss, um sich an die durch die Rekonfigurationen ändernden Eigenschaften anzupassen. Mögliche Anpassungen wären beispielsweise die Verkürzung der Abarbeitungszeiten spezieller Operatoren, oder die Anpassung der Abarbeitungsreihenfolge der einzelnen Befehle, da gewisse Operatoren zu bestimmten Zeitpunkten nicht verfügbar sind. Hierbei muss sichergestellt werden, das die Funktionalität des Programms durch die Änderungen der Befehlsabfolge erhalten bleibt.
Innerhalb dieser Arbeit wird zum besseren Verständnis für den Leser, als Assemblercode eine lineare, sequenzielle Abfolge von lesbaren Assemblerbefehlen verstanden. Weiterhin wird unter Maschinencode die durch einen Assembler übersetzte, sich potenziell in der Abfolge der Befehle unterscheidende, sequenzielle Binärcodeabfolge mit zusätzlichen innerhalb des Codes codierten Informationen verstanden. Am Ende der Verarbeitungskette steht ein speziell auf ein Projekt hin optimierter Softcore Prozessor mit einer aufgabenspezifischen Befehlsabfolge in Form eines

für den Softcore lesbaren Maschinencodes und ein optionaler Rekonfigurationsplan für den Prozessor.
Anhand dieser Verarbeitungskette werden in den folgenden Abschnitten 4.4 bis 4.6 die jeweiligen Verarbeitungsschritte, die in Abbildung 4.3 aufgezeigt worden, einzeln erläutert:

- Abschnitt 4.4, befasst sich mit verschiedenen Konzepten zur Erstellung, Erweiterung und Wiederverwendung von Softcore Prozessoren in der Domäne der harten Echtzeit. Nach einem allgemeinen Aufbau möglicher Softcores wird eine Erweiterung vorgestellt, deren Ziel eine Reduktion des Energiebedarfs ist. Notwendige Anpassungen, wie Kommunikationsstrukturen bei Mehrkernarchitekturen sowie Konzepte zur partiellen Rekonfiguration von Softcore Prozessoren werden diskutiert.

- Der zweite Abschnitt, Abschnitt 4.5 befasst sich mit der Methodik, automatisiert Assemblercode aus einem Datenflussgraphen zu erzeugen (*Dataflow Model* → *Model-to-Code Generator* → *Assembly Code*).
  Verschiedene graphbasierte Optimierungsverfahren sind dabei ein wesentlicher Bestandteil. Ein weiterer wichtiger Punkt ist die Minimierung des Abstraktionsoverheads. Hier wird ein Verfahren erläutert, dass die benötigte Menge an Speicher des resultierenden Assemblercodes minimiert, um eine optimale Übergabe zu (potenziell) verschiedenen optimierenden Assemblern mit unterschiedlichem Funktionsumfang zu ermöglichen.

- In Abschnitt 4.6 werden Methoden vorgestellt, wie Assemblercode gescheduled werden kann, d. h. wie aus einem sequenziellen Assemblercode ein verschiedene Pipelines verwendender Maschinencode erzeugt werden kann (*Assembly Code* → *Assembler* → *Softcore Machine Code*). Dabei wird in jedem vorgestellten Verfahren immer die Einschränkung der harten Echtzeitfähigkeit und notwendige Anpassungen entsprechender Strategien diskutiert. Ein besonderes Augenmerk liegt auf verschiedenen Optimierungsstrategien wie spezieller Speicherverwaltung sowie auf optimierenden Schedulingstrategien, die eine Maximierung der Pipelineauslastung zum Ziel haben.

## 4.4 Konzept eines echtzeitfähigen Softcore Prozessors

### 4.4.1 Allgemeiner Aufbau

Der wesentliche Schritt, der in Abschnitt 4.3 unter Abbildung 4.3 vorgestellten Toolchain, ist die Definition und Umsetzung eines spezialisierten Softcore Prozessors aus einer Softcore-Bibliothek. Dazu muss der Softcore in wesentlichen Teilen bereits vordefiniert vorliegen. Besonders die Rahmenbedingungen, also die Anforderungen und Einschränkungen, müssen vordefiniert sein und dürfen sich nicht verändern. Hierzu zählen vorgegebene Eigenschaften, die sich aus der Aufgabendomäne ableiten sowie Einschränkungen die die Kompatibilität mit den anderen Teilen der Entwicklungskette sicherstellen. Der allgemeine Aufbau eines Softcore Prozessors unterscheidet sich dabei nicht von einem normalen Prozessor und ist in Abbildung 4.4 dargestellt.

Dabei besteht der Softcore aus einem Programmspeicher (*Program Memory*), einem zugehörigen Befehlsdekodierer (*Instruction Decoder*), einem Datenspeicher (*Data Memory*) sowie einem Rechenwerk (*ALU*) sowie einer Möglichkeit mit der umgebenden Logik kommunizieren zu können (*I/O Register*).

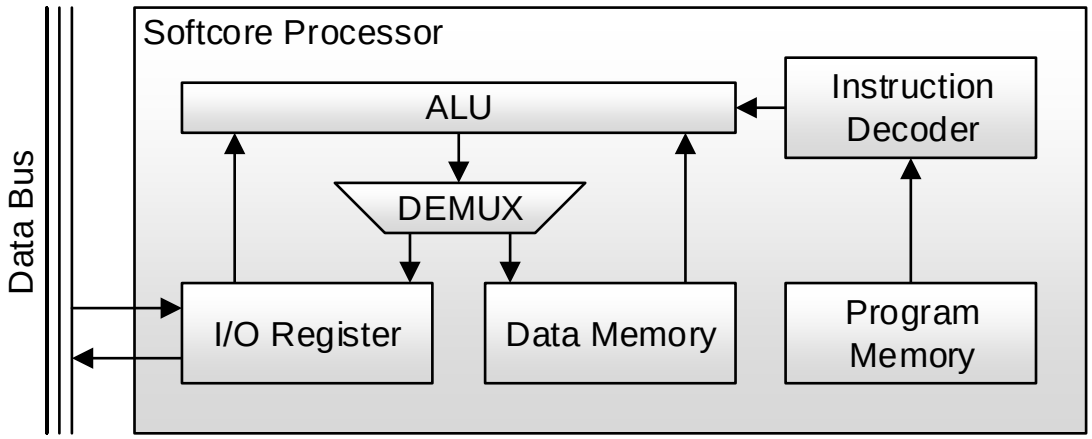



Abbildung 4.4: Allgemeiner Aufbau eines Softcore Prozessors

Im Folgenden wird auf die wesentlichen Teile und deren Anpassungen an die Aufgabendomäne im Detail eingegangen:

#### Programmcode

Um die Eigenschaft an harte Echtzeitfähigkeit bereits zur Designzeit nachweisen und sicherstellen zu können, müssen einige Einschränkungen vorgenommen werden. Diese betreffen vor allem den Programmcode. So ist es nur in hohen Abstraktionsebenen erlaubt, Schleifen zu verwenden. Diese müssen Eigenschaften besitzen, die es ermöglichen die Schleifen bereits zur Designzeit auflösen zu können. Das bedeutet, sie müssen beispielsweise eine vorgegebene Anzahl an Durchläufen besitzen und können keine durch andere Bedingungen variierenden Abbruchbedingungen besitzen. Diese Schleifen sind nur in der Modellabstraktionsebene erlaubt und werden spätestens vom Assembler beim Übersetzungsvorgang in Maschinencode aufgelöst.
Eine zusätzliche notwendige Einschränkung ist das Verbot von bedingten Sprüngen innerhalb des Assemblercodes. Bedingte Sprünge innerhalb des Maschindencodes zur Laufzeit können unvorhersehbare Auswirkungen auf die Laufzeit von einzelnen Programmteilen haben und würden eine vorherige zeitliche Analyse unmöglich machen.
Diese Einschränkung ist in der adressierten Aufgabendomäne allerdings ein unwesentlicher Nachteil, da die hier realisierten Aufgaben zur Designzeit feststehen und damit Mechanismen umgesetzt werden können, die entsprechende Sprünge ersetzen, sich aber dennoch nicht negativ auf die Analysierbarkeit des Programmcodes auswirken. Da die meisten Sprünge Schleifen umsetzen, können diese entsprechend ersetzt werden. Es kann eine Hardwareschleife eingesetzt werden. Weiterhin besteht die Möglichkeit, eine Schleife im Quellcode auszurollen.

#### Daten- und Programmspeicher

Da der Softcore auf einem FPGA betrieben werden soll, muss zur Designzeit eine Definition der benötigten Ressourcen für die Daten- und Programmspeicher erfolgen. Diese Informationen sind

programmspezifisch, ändern sich also mit jedem Assemblercode, und werden vom Assembler bereitgestellt. Beim Übersetzungsvorgang wird durch den Assembler genau ermittelt wie viel Speicher der Programmcode benötigt und wieviele Variablen und Konstanten in den Speicherzellen abgespeichert werden müssen.

#### Arithmetic Logic Unit (ALU)

Zur Berechnung von hochpräzisen applikationsspezifischen Bildverarbeitungalgorithmen oder Regelalgorithmen werden Gleitkommaberechnungen ausgeführt, die eine entsprechende Genauigkeit besitzen müssen. Aus diesem Grund sollte ein skalierbarer Floating-Point Standard mit einer Länge von mindestens 32 Bit verwendet werden. Vorgeschlagen wird der in [IEE85] definierte Standard.
Aus Gründen der besseren Analysierbarkeit des Programmcodes muss die ALU auch entsprechend angepasst werden. Deswegen soll für die in der echtzeitkritischen Daten- und Bildverarbeitung verwendeten Softcore Prozessoren eine Harvard-RISC-Architektur [PH17] mit einer in Abbildung 4.5 dargestellten fünfstufigen Pipeline verwendet werden. Um eine schnelle


Abbildung 4.5: Fünfstufige Pipeline

Abarbeitung des Maschinencodes zu gewährleisten, sollte jeder in der ALU verwendete Operator intern ebenfalls eine Pipeline besitzen. Das bedeutet, dass beispielsweise bei einer Addition, die drei Takte für eine Berechnung benötigt, theoretisch drei Takte nach dem ersten Start in jedem Takt ein Ergebnis erzeugt wird.
Die innerhalb der ALU verwendeten Operatoren sollten sich an die Aufgabendomäne anpassen und im Idealfall zur Designzeit austauschbar gestaltet werden. Einen besonderen Stellenwert nehmen hier vergleichsweise komplexe Operatoren ein. So ist es beispielsweise sinnvoll, Exponentialfunktionen, Logarithmusfunktionen oder vergleichbare Funktionen nativ in einem eigens dafür vorgesehenen Operator berechnen zu können, um eine zeiteffiziente Programmverarbeitung zu ermöglichen und die gesetzten Echtzeitschranken einzuhalten. Auch sogenannte kombinierte Operatoren sind denkbar. Das sind Operatoren, die innerhalb eines einzelnen Befehls multiple Berechnungen durchführen, also eine festgelegte frequentiert benötigte Abfolge von einzelnen Operationen.

### 4.4.2 Konzept zur Verringerung des Energieverbrauchs

Entsprechende Erweiterungen des Softcore Prozessors leiten sich aus der Aufgabendomäne ab, so ist es beispielsweise notwendig, im Einsatz von vielen eingebetteten Regelsystemen den benötigten Energiebedarf zu minimieren, da diese Systeme nur über einen begrenzen Energievorrat verfügen. Entsprechend wird die Möglichkeit geschaffen, nicht benötigte Teile der Hardware zeitweise nicht mit Strom zu versorgen. Hierzu zählen vor allem Teile der ALU und der Datenspeicher. Sollte im Rahmen des Starts einer neuen Operation ein Teil des Datenspeichers

nicht benötigt werden, oder wird im aktuellen Takt keine neue Operation gestartet, dann kann die Taktversorgung zum Datenspeicher zu diesen Zeitpunkten komplett abgeschaltet werden. Dasselbe gilt für einzelne Operatoren der ALU: sollten diese im aktuellen Programmabschnitt nicht zu Berechnungen verwendet werden, können diese abgeschaltet werden. Daraus ergibt sich eine Anpassung der allgemeinen Hardwarestruktur:

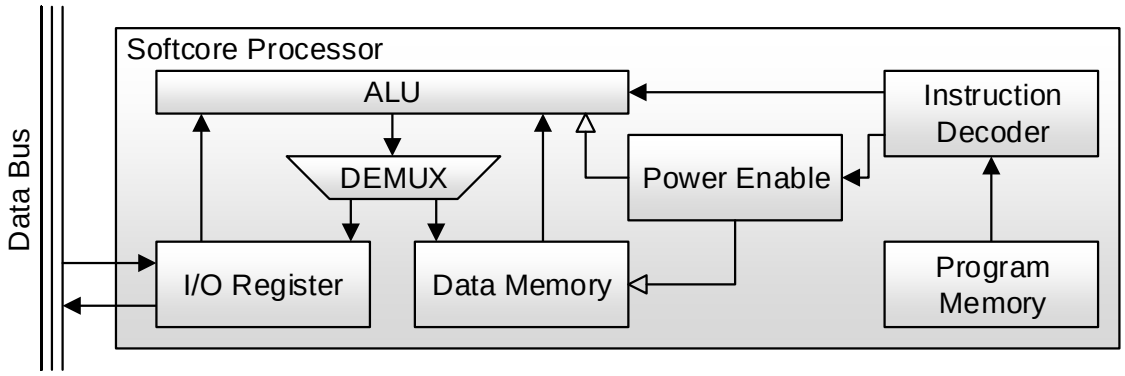



Abbildung 4.6: Softcore Prozessor Erweiterung

### 4.4.3 Konzepte zur Erweiterung zu einer echtzeitfähigen Mehrkernarchitektur

Im Folgenden werden verschiedene Konzepte zur Erweiterung der Softcore Architektur auf eine Mehrkernarchitektur diskutiert. Ein Trend der in allen Prozessorarchitektur-Bereichen zu beobachten ist, ist die Verwendung von Mehrkernarchitekturen. Mehrkernprozessoren können dabei mehrere Befehlsstränge parallel ausführen. Das führt zu einer Reihe von Vorteilen und Herausforderungen, die im weiteren kurz diskutiert werden:
Ein Problem bei der Erweiterung von Softcore Prozessoren zu Mehrkernarchitekturen ist die Notwendigkeit, Befehlsstränge synchronisieren zu müssen und der Austausch von Daten zwischen den Prozessorkernen. Hierzu gibt es eine Vielzahl an Lösungsansätzen, die sich in Komplexität, nutzbarer Bandbreite, Latenz, Skalierbarkeit und Ressourcenbedarf (auf dem FPGA) unterscheiden.

#### Gemeinsamer Speicher

Eine verbreitete Möglichkeit zum Datenaustausch zwischen verschiedenen Prozessorkernen ist die Verwendung von einem gemeinsamen Speicher (*Shared Memory*). Dabei sollte das ein zentraler Speicher sein, der von allen Prozessorkernen verwendet werden kann. Jeder Prozessorkern verwendet seinen exklusiven Speicher und der gemeinsame Speicher dient zum Weitergeben und Nutzen gemeinsam benötigter Ergebnisse. Wichtig ist zu beachten, dass dieser Teil des Prozessors Auswirkungen auf die maximale Taktrate haben kann, da mehrere Kerne an den Speicher angeschlossen sein müssen und dadurch sichergestellt werden sollte, dass die Prozessorkerne physisch auf dem FPGA möglichst nah an diesem Speicher platziert werden, da die Laufzeiten der ansonsten entstehenden langen Leitungen sich negativ auf die maximale Taktrate auswirken kann.
Größe und Bandbreite des Speichers müssen an die Anzahl an verwendeten Prozessorkernen

angepasst werden und es muss sichergestellt werden, dass es im Einsatz nicht zu Inkonsistenzen durch die gleichzeitige, multiple Verwendung der gleichen physischen Speicheradresse kommen kann. Hierzu können Flag-Bits verwendet werden, oder das „exclusive-Cache“-Prinzip, bei dem die Daten bei Benutzung aus dem gemeinsamen Speicher entfernt werden. Auch eine Verwendung von Multimasterbussen ist denkbar. [HMN09]
Wollen mehrere Kerne im selben Takt auf einen Speicherslot zugreifen, müssen Mechanismen zur Priorisierung, oder feste Zeitslots verwendet werden. Eine weitere Möglichkeit wäre die Verwendung von redundantem Speicher, der einen zeitlichen mehrfach Zugriff erlaubt.

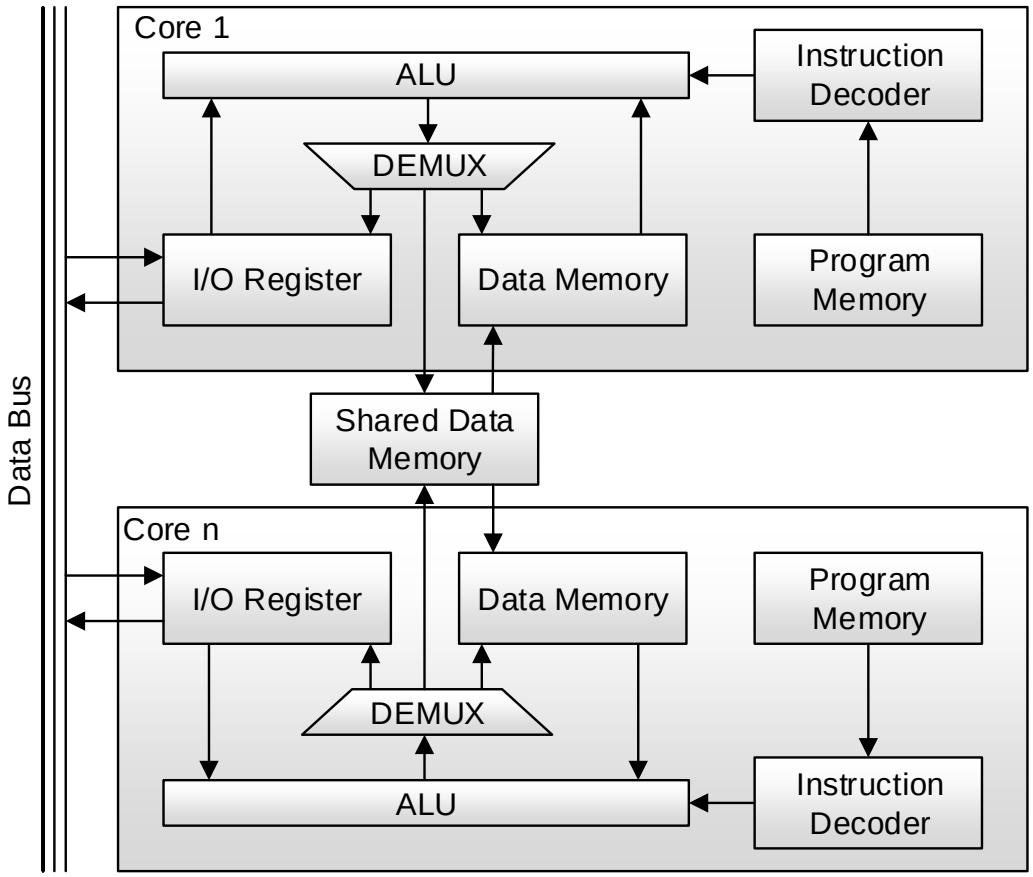



Abbildung 4.7: Mehrkernarchitektur mit gemeinsamen Speicher

### Synchronisation über Nachrichten

Eine weitere Möglichkeit mehrere Prozessorkerne zu synchronisieren, ohne einen gemeinsamen Speicher zu verwenden, ist der Austausch von Informationen über Nachrichten. Ein entstehender Vorteil bei der Verwendung von nachrichtenbasierten Synchronisationsmechanismen ist die Weiterleitung von Nachrichten. Wenn Nachrichten über mehrere Kerne hinweg weitergeleitet werden, ist eine indirekte Kommunikation möglich und das verbessert die Skalierbarkeit der Mehrkernarchitektur. Die Kommunikation über Nachrichten bedeutet dabei immer einen zusätzlichen Overhead, da neben den eigentlichen Nutzdaten auch die Kommunikationsdaten

mit übertragen werden müssen. Anstatt direkt auf die Daten eines anderen Kerns über den gemeinsamen Speicher zugreifen zu können, müssen die Daten über eine Nachricht angefordert werden. Das führt zu einer erhöhten Latenz beim Datenaustausch. [YYX+12]

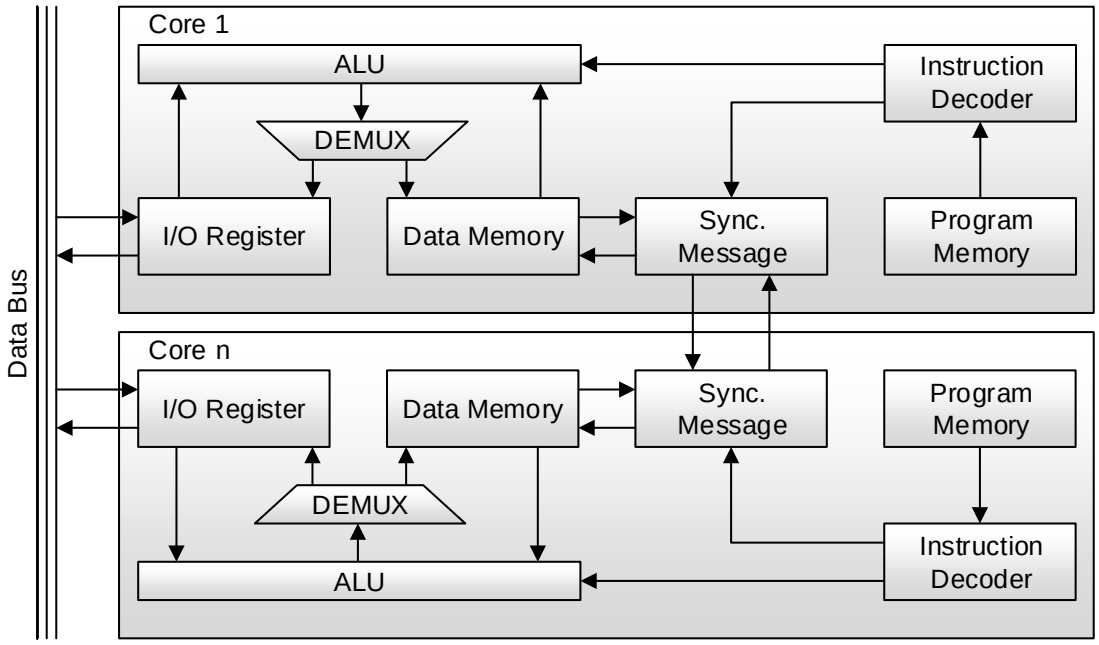



Abbildung 4.8: Nachrichtenbasierte Mehrkernarchitektur

### Verbindungstopologien

Sollen mehr als zwei Prozessorkerne miteinander verbunden werden, muss eine passende Verbindungsstruktur umgesetzt werden. Da eine direkte Verbindung zwischen allen Kernen exponentiellen Aufwand bedeutet, müssen alternative Strukturen diskutiert werden. Dabei reagieren unterschiedliche Umsetzungen verschieden auf die Erhöhung der Anzahl an Prozessorkernen. Die unterschiedlichen Verbindungstopologien sind dabei sowohl für gemeinsamen Speicher, als auch für nachrichtenbasierte Kommunikation denkbar. [New08]

- Busbasierte Verbindungstopologien
  Wird die Verbindungstopologie über ein Bussystem aufgebaut, so nutzen alle Prozessorkerne diesen als zentralen Kommunikations- bzw. Datenkanal.

  Der Vorteil bei dieser Verbindungsart liegt in der Tatsache, dass unabhängig von der Anzahl an verwendeten Kernen jeder Kern in der Lage ist, direkt mit jedem anderen Kern zu kommunizieren oder Daten auszutauschen. Daraus ergibt sich auch direkt ein Nachteil: die durch den Bus zur Verfügung stehende Bandbreite müssen sich alle Kerne untereinander teilen. Weiterhin müssen mögliche Kollisionen verhindert werden.
  Durch die Mechanismen zur Kollisionsvermeidung und Aufteilung der Bandbreite erhöht sich die durchschnittliche Latenz. Eine weitere Folge ist die steigende Länge von Verbindungsleitungen mit steigender Anzahl an Prozessorkernen, was eine Senkung der jeweiligen

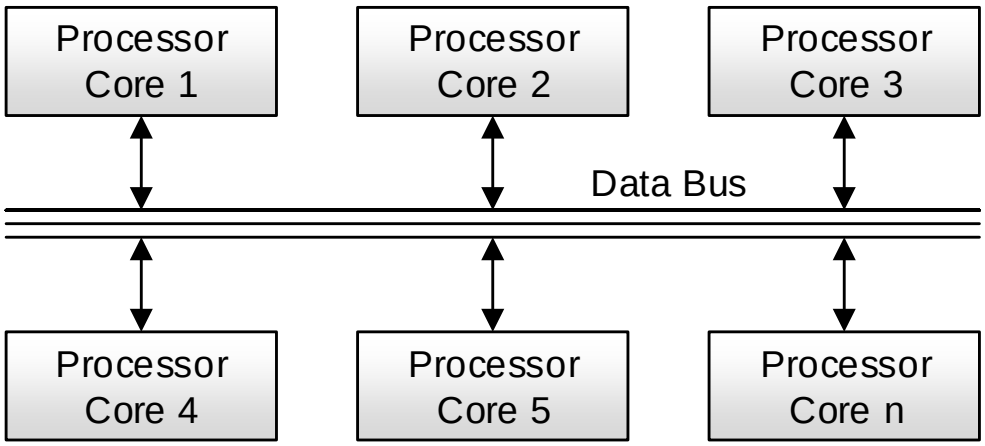



Abbildung 4.9: Busbasierte Verbindungstopologie

maximalen Taktrate zur Folge haben kann. Da die Bandbreite des Bussystems meist nicht veränderbar ist, kann das System nicht mit einer beliebig hohen Anzahl an Kernen skalieren. [BDM09, New08]

- Sternbasiertebasierte Verbindungstopologien
  Bei einer sternbasiertebasierten Verbindungstopologie wird der Datenaustausch aller Kerne über einen zentralen Prozessorkern realisiert. Dieser kann dabei nur vermindert oder keine anderen Berechnungen durchführen.

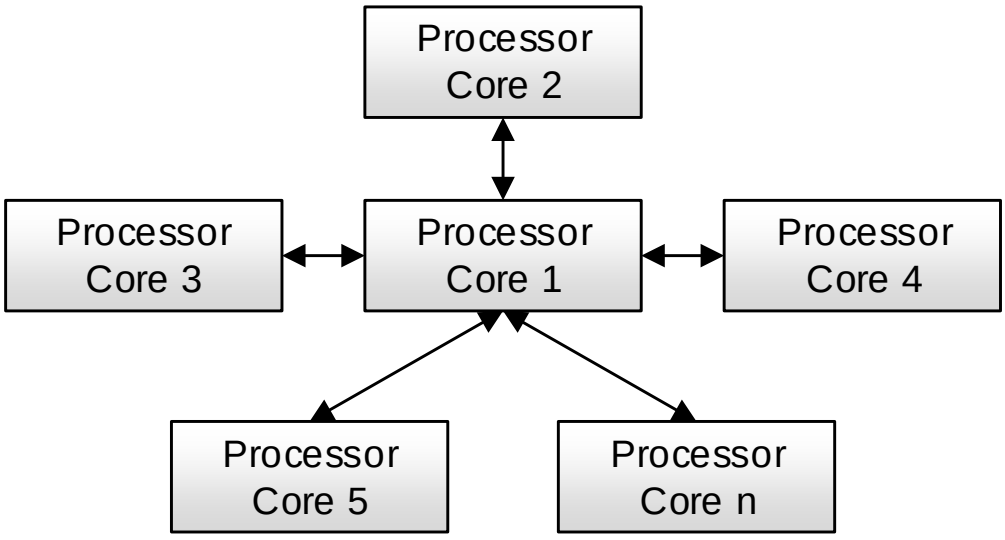



Abbildung 4.10: Sternbasierte Verbindungstopologie

Soll lediglich ein reines Austauschen von Nutzdaten über diesen zentralen Knotenpunkt erfolgen, also keine weitere Nachrichtenvermittlungslogik, kann dieser auch von einem kompletten Prozessorkern auf einen gemeinsam genutzten Speicher reduziert werden. Der Vorteil liegt in den kurzen Verbindungsleitungen, da jeder Prozessorkern nur mit dem zentralen Kern verbunden werden muss. Dadurch werden höhere Taktraten im Vergleich zu einer Busverbindung ermöglicht. Ein Nachteil liegt in dem zu behandelnden Fall, dass mehrere Kerne mit exakt einem weiteren Kern gleichzeitig kommunizieren wollen, hier muss eine Priorisierungslokik, bzw. eine Pufferlogik der Nachrichten vorgesehen werden. Sollte lediglich ein gemeinsamer Speicher genutzt werden, muss ein Algorithmus zur Konfliktlösung bei gleichzeitigem Zugriff auf dieselbe Speicherzelle umgesetzt werden. Diese Konfliktfälle schränken eine entsprechende Skalierbarkeit ein, da diese Fälle mit steigender Kernzahl zunehmen, was den gesamten Durchsatz jedes einzelnen Kerns einschränkt. [BDM09, New08]

- Ringbasierte Verbindungstopologien
  Die Idee hinter einer Ringbasierten Verbindungstopologie liegt darin, immer die jeweils benachbarten Prozessorkerne miteinander zu verbinden. Dabei wird davon ausgegangen, das der erste und der letzte Kern benachbart sind.

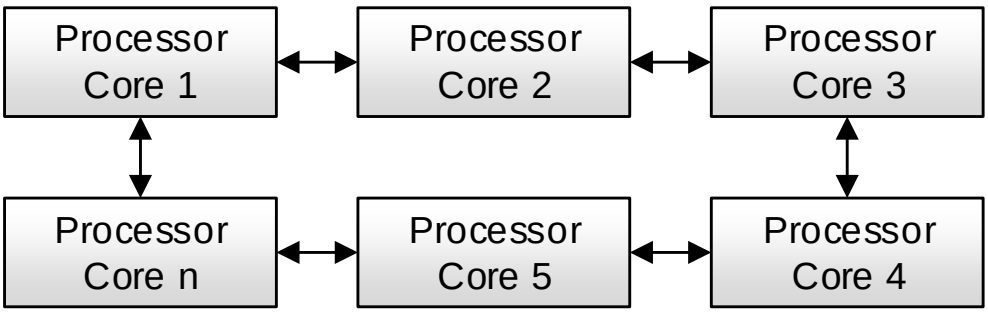



Abbildung 4.11: Ringbasierte Verbindungstopologie

Hierbei werden Nachrichten an den jeweiligen Nachbar weitergeleitet, bis diese ihren Zielprozessorkern erreicht haben. Hierbei wird die Daten- bzw. Nachrichtenlaufzeit mit steigender Kernzahl erhöht, da diese Daten im Mittel einen längeren Weg zurücklegen müssen. Da allerdings jeder Kern nur mit seinem direkten Nachbarn kommunizieren muss, ist es möglich sehr kurze Verbindungsleitungen zu verwenden, die eine hohe Geschwindigkeit ermöglichen. [BDM09, New08]

- Zweidimensionale Verbindungstopologien
  Wird die ringbasierte Verbindungstopologie zu einer zweidimensionalen Gitterstruktur erweitert, erhält man die hier vorgestellte Verbindungstopologie. Dabei werden voneinander unabhängige Ringverbindungen sowohl in allen Zeilen, als auch in allen Spalten innerhalb der Gitterstruktur aufgebaut. Dadurch ist jeder Prozessorkern mit exakt zwei Ringverbindungen verbunden. Dabei können Nachrichten von einem Ring in einen anderen Ring weitergeleitet werden. Aufgrund der hohen Komplexität an Verbindungen und möglichen Daten- und Nachrichtenwegen, ist es notwendig einen Routingalgorithmus zu realisieren. Die Skalierbarkeit ist dabei besser, als die einer reinen ringbasierten

Verbindungstopologie, da die auftretenden Wege vergleichsweise kurz bleiben. [BDM09, New08]
Es sei erwähnt, dass dieses Konzept theoretisch auf n-Dimensionen erweiterbar ist. Das ist allerdings im Rahmen dieser Arbeit nicht von Interesse, da davon ausgegangen wird, das sich die tatsächliche Anzahl an Kernen in einer Multikernarchitektur im kleineren einstelligen Bereich bewegt.

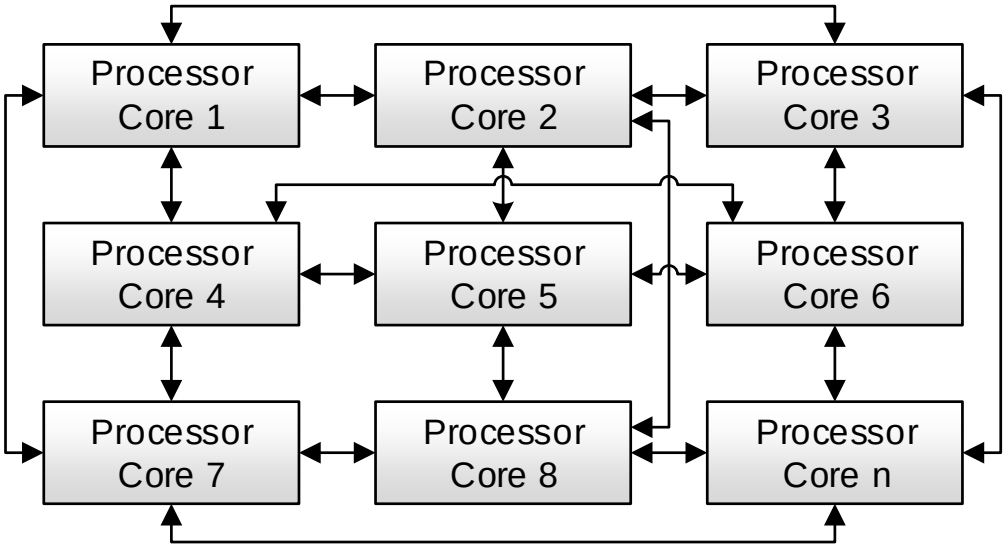



Abbildung 4.12: Zweidimensionale Verbindungstopologie

#### Ressourcenreduktion bei Mehrkernarchitekturen

Einer der wesentlichen Vorteile bei der Verwendung von Mehrkernarchitekturen ist die Möglichkeit, ressourcenintensive Operatoren nur auf einem oder wenigen der verwendeten Prozessorkerne verfügbar zu machen. Sinnvoll ist dieser Ansatz immer dann, wenn entsprechende Operatoren nicht frequentiert verwendet werden. Das Ergebnis ist eine heterogene Architektur, wie beispielsweise in [SKV+06] beschrieben.
Es muss allerdings unterschieden werden, ob ein Operator als eine exklusive Ressource für mehrere Prozessorkerne zur Verfügung steht, oder ob dieser Operator nur auf einem Kern verfügbar ist. Der Unterschied liegt darin, dass bei der Verwendung als exklusive Ressource für alle Kerne theoretisch jeder Kern diese Berechnung durchführen kann, währenddessen im anderen Fall diese Operation nur von einem spezifischen Kern ausgeführt werden kann. Im ersten Fall muss eine Priorisierungsfunktion entscheiden, wann welcher Prozessor die Berechnungen durchführen darf, während bei der zweiten Version der Assembler dafür sorgen muss, dass entsprechende Operationen nur exklusiv auf einem Kern ausgeführt werden und dass alle dafür notwendigen Argumente zum Zeitpunkt der Berechnung dem Kern vorliegen. Durch das Anfordern auf dem Kern nicht verügbarer Argumente entsteht ein Kommunikationsaufwand an Steuerinformationen mit anderen Kernen, da diese Daten angefordert werden müssen bevor sie übertragen und genutzt werden können.
Es tritt also entweder ein erhöhter Kommunikationsaufwand oder ein (potenziell) erhöhter

Datenaustausch zwischen den Kernen auf.
Hier wird auch der Vorteil der Festlegung sämtlicher Operationen auf entsprechende Kerne zur Designzeit deutlich: ein Kommunikationsaufwand zur Laufzeit entfällt und ein notwendiger und zeitkostender Datenaustausch wird minimiert.

### 4.4.4 Konzept zur partiellen Rekonfiguration von spezialisierten Softcore Prozessoren

Eine sinnvolle Erweiterung der Funktionalität eines Softcore Prozessors, der im Bereich der Bildverarbeitung eingesetzt wird, ist die Möglichkeit Teile des Prozessors oder sogar ganze Prozessoren dynamisch zur Laufzeit austauschen zu können. Bei der Bildverarbeitung treten oft sich über die Zeit stark ändernde Anforderungen auf, beispielsweise ein kompletter Wechsel des zugrunde liegenden Verarbeitungsalgorithmus. Als Beispiel sei auf das in Abschnitt 2.6.2 vorgestellte Verfahren der AutoVision SoC Architektur verwiesen. Hier werden während der Autofahrt von einer Kamera Bilder erzeugt. Diese Bilder müssen von einem System mit Rekonfigurierbaren Co-Prozessoren verarbeitet werden. Da die Anforderungen über die Zeit stark schwanken, (z. B. Fahrt bei Sonnenschein und Fahrt in einem Tunnel) muss sich das System daran anpassen können. Diese Anpassung wird über die Rekonfiguration der Co-Prozessoren gelöst. Die harte Echtzeitschranke beträgt dabei 32,25ms für die Rekonfiguration mit anschließender Bildverarbeitung je Bild. In der Arbeit [Cla11] ist dieses Szenario untersucht worden und der Umgang in der praktischen Anwendung wurde ausführlich erläutert.
In solchen Fällen werden oft in den Algorithmen Operatoren zur Berechnung eingesetzt, die zum einen sehr viele physische Ressourcen auf dem FPGA benötigen, der neue Algorithmus aber andere sehr ressourcenlastige Operationen benötigt. In diesen Fällen ist es sinnvoll, entsprechende Operatoren bei einem Wechsel des Verarbeitungsalgorithmus ebenfalls gegeneinander auszutauschen.

#### Voraussetzungen zur Anwendung der partiellen Rekonfiguration

Die grundlegende Voraussetzung zur Anwendung der partiellen Rekonfiguration ist die Verwendung einer geeigneten Hardwareplattform. Wie in Abschnitt 2.4.2 definiert wurde, wird unter partieller Rekonfiguration die Fähigkeit von FPGAs verstanden, Teile der Chipfläche zur Laufzeit zu rekonfigurieren. Diese Voraussetzung schränkt die Verwendbarkeit der FPGA-Familien und Hersteller auf genau die FPGAs ein, die diese Eigenschaft besitzen. Als Beispiele können hier exemplarisch die Arria 10 Familie der Firma Intel (ehemals Altera), oder die 7-Series der Firma Xilinx genannt werden.
Optimierungsalgorithmen, welche die Ressourcen oder die Platzierung auf dem Chip optimieren, dürfen nicht über die festgesetzten Grenzen der statischen und rekonfigurierbaren Blöcke optimieren. Das bedeutet, dass entsprechende Optimierungsalgorithmen separiert für statische und für jede rekonfigurierbare Blöcke eingesetzt werden müssen.
Entsprechend dieser Eigenschaft dürfen ebenfalls keine Logikelemente von rekonfigurierbaren Blöcken in die Region des FPGAs, auf der sich die statischen Blöcke befinden, gesetzt werden. Es muss sichergestellt sein, dass die gesamten Ressourcen einer festgesetzten Region für regkonfigurierbare Blöcke auch ausschließlich für diese verwendet werden.
Die Partition-Pins, also die Pins an den jeweiligen Blockgrenzen, dürfen sich zwischen unterschiedlichen Konfigurationen des gleichen rekonfigurierbaren Blockes nicht verändern. Das

würde zu einem inkompatiblen Design führen und unvorhersehbare Effekte erzeugen.
Es ist möglich, Routen eines statischen Designs durch die für rekonfigurierbare Blöcke reservierten FPGA Chipflächen zu routen. Dies unterliegt allerdings der gleichen Einschränkung wie bei den Partition-Pins: sobald das Routing in einem Design festgelegt wurde, muss dieses für alle weiteren in exakt derselben Weise erfolgen. Diese statischen Routen durch rekonfigurierbare Blöcke werden als Feed-Through Routes [RFG16] bezeichnet. Das Routing solcher Routen darf dabei nicht mehrere rekonfigurierbare Regionen verwenden.
In der Aufgabendomäne, in der ein in dieser Arbeit adressierter Softcore Prozessor zum Einsatz kommt, muss sichergestellt werden, das die zeitlichen Kosten eines Austauschs der Hardware nicht zu einer Verletzung der Echtzeitschranke führen.
Dabei kann bei jedem Projekt eine theoretische Vorbetrachtung angefertigt werden, ob eine Rekonfiguration in dem jeweiligen Projekt sinnvoll ist oder nicht. Die Vorbetrachtung ist in diesem Fall eine Berechnung, wie viel Zeit eine Rekonfiguration voraussichtlich benötigen wird. Dazu muss die jeweilige FPGA-Familie sowie die Größe der zu Rekonfigurierenden Region, bzw. der darin enthaltenen Logik, bekannt sein. Die für die Berechnung relevanten Daten sind in Tabelle 4.1 dargestellt.

| Komponente | Größe (Bit) | zusätzliche Informationen |
|---|---|---|
| Header | 960 | Busbreite Pattern, ID-Code-Check, Synchronisationsinformationen |
| Schreibbefehl | 256 | |
| Endsequenz | 736 | CRC, NOPs |
| Konfigurationsframe | 3232 | Größe je Frame |

Tabelle 4.1: Größe der Komponenten partieller Bitstreams

Die Werte beziehen sich auf die Zynq-7 Familie und können bei anderen FPGA-Familien abweichen. Anhand dieser Werte kann abgeschätzt werden, wie viel Zeit eine potenzielle Rekonfiguration benötigen würde:
Deutlich wird das am folgenden Beispiel. Es wird davon ausgegangen, dass eine Region mit insgesamt 100 Konfigurationsframes rekonfiguriert werden soll. Diese Region befindet sich dabei in einer einzigen Clock-Region und ist ein zusammenhängender Block. Das bedeutet, es gibt keine Sprünge in der Frameadresse. Dadurch ergibt sich folgende Rechnung:

$$\text{Bitstreamgröße} = \text{Header} + 1 \cdot \text{Schreibbefehl} + 100 \cdot \text{Konfigurationsframes} + 1 \cdot \text{Endsequenz} \tag{4.1}$$

Das resultiert in einer Bitstreamgröße von 325.152 Bit. Es ist zu beachten, dass für jeden Sprung in der Frameadresse, z. B. bei Wechsel der Clock-Region weitere 256 Bit für zusätzliche Schreibbefehle hinzukommen. Mit der Bitstreamgröße und den Informationen über die Geschwindigkeit des Rekonfigurationsports lässt sich die benötigte Rekonfigurationszeit abschätzen. In diesem Beispiel wird angenommen, dass der Rekonfigurationsport bei einer Frequenz von 100 MHz einen theoretisch maximalen Durchsatz von 400 MB/s bewältigt. Daraus ergibt sich, dass die Rekonfiguration eine Rekonfigurationszeit von mindestens 812,88 $\mu$s benötigen würde. In der Praxis ist die tatsächliche Rekonfigurationszeit zusätzlich abhängig vom maximalen Delay und Durchsatz des verwendeten Bitstream-Speichers. Dieser muss zusätzlich in Betracht gezogen werden. Diese Berechnung stellt damit eine untere Schranke dar, das bedeutet eine Rekonfiguration wird niemals in kürzerer Zeit erfolgen.
Mit Hilfe dieser Informationen kann abgeschätzt werden, ob eine partielle Rekonfiguration zur

Verletzung der Echtzeitschranke führen würde. Sollte bereits dieser theoretisch berechnete Wert über der Echtzeitschranke liegen, kann keine Rekonfiguration umgesetzt werden. Sollte dies nicht der Fall sein, wird der Vorteil der Verwendung deutlich: Die Berechnungen inklusive des zeitlichen Rekonfigurationsoverheads sind schnell genug um die Echtzeitkriterien zu erfüllen, aber dank der Rekonfiguration können Logikressourcen eingespart werden und damit können kleinere und dadurch billigere FPGAs verwendet werden.

### Rekonfigurationsgranularität bei Softcore Prozessoren

Wie bereits in Abschnitt 2.4.2 allgemeingültig für alle FPGAs mit der Fähigkeit zur partiellen Rekonfiguration vorgestellt wurde, wird eine Einteilung der rekonfigurierbaren Bausteine nach ihrer Granularität vorgenommen, mit der diese die Rekonfiguration durchführen können. Diese Einteilung wird nun auf Softcore Prozessoren spezialisiert. Hierbei wird davon ausgegangen, dass nach dem in Abschnitt 2.4.2 vorgestellten Schema jede im Weiteren vorgestellte Rekonfiguration als Rekonfiguration mit grober Granularität eingeteilt wird, da in jedem Fall ganze Funktionsblöcke ersetzt werden.
Dabei wird allerdings unterschieden, auf welcher Ebene die Rekonfiguration stattfindet.

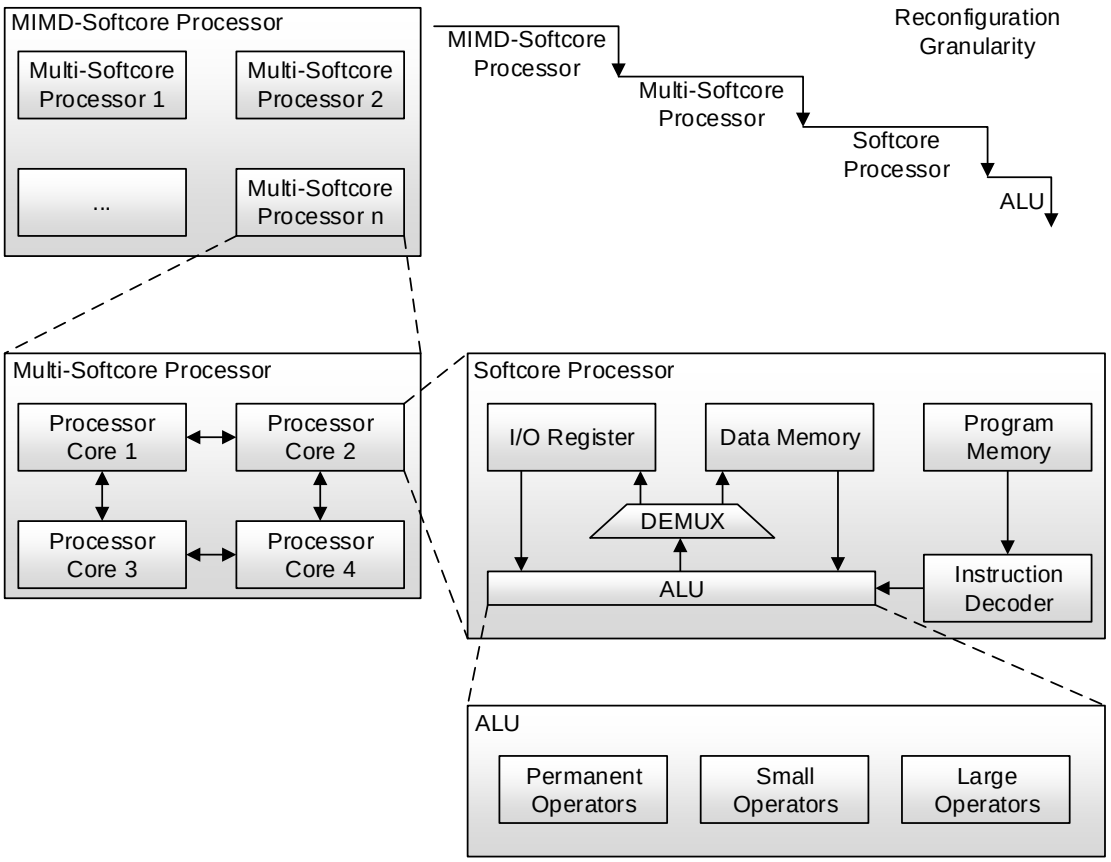



Abbildung 4.13: Rekonfigurationsgranularität von Softcore Prozessoren

Wie in Abbildung 4.13 illustriert wird, ist es im Einsatzgebiet der Bildverarbeitung sinnvoll, MIMD-Prozessoren (*Multiple Instruction Multiple Data-Softcore Processor* (*MIMD-Softcore*

*Processor*)) zur parallelen Auswertung von beispielsweise einzelnen Bildpunkten einzusetzen, um die Verarbeitungsgeschwindigkeit zu maximieren. Ein MIMD-Softcore Prozessor ist dabei aus einzelnen Softcore Prozessoren aufgebaut. Je nach Anwendungsszenario, können diese mit einer Mehrkernarchitektur (*Multi-Softcore Processor*) ausgestattet sein. Dabei ist anzumerken, dass von der Ebene der MIMD-Softcore Prozessoren aus betrachtet, jeder interne Softcore Prozessor unabhängig arbeitet. Allerdings ab der Ebene der Multi-Softcore Prozessoren, führen diese ein vorgegebenes Programm parallel und synchron aus, um die bereits in dieser Arbeit genannten Vorteile ausnutzen zu können. Auf der Ebene des Multi-Softcore Prozessors agiert dieser als VLIW-Prozessor.
In Abbildung 4.13 zu sehen ist exemplarisch eine Mehrkernarchitektur (*Multi-Softcore Processor*) mit vier einzelnen Prozessorkernen (*Processor Core*), die mittels einer, wie in Abschnitt 4.4.3 vorgestellten, ringbasierten Verbindungstopologie miteinander kommunizieren. Jeder Softcore Prozessor (*Softcore Processor*) ist dabei nach der unter Abschnitt 4.4.1 vorgestellten Architektur aufgebaut.
Bei der Rekonfigurationsgranularität von Softcore Prozessoren werden nun die verschiedenen Ebenen unterschieden, auf denen rekonfiguriert werden kann. Die oberste Ebene, und damit die gröbste Granularität, wird erreicht, wenn ganze Prozessoren innerhalb eines MIMD-Prozessors ausgetauscht werden. Da bei dieser Rekonfiguration physisch viele Konfigurationssektionen angepasst werden müssen, ist diese Rekonfigurationsart vergleichsweise zeitintensiv.
Eine Ebene tiefer werden innerhalb von Mehrkern-Softcore Prozessoren einzelne Prozessorkerne ausgetauscht. Auf dieser Ebene ist es auch möglich, den Prozessor bei einem Algorithmenwechsel entsprechend mit neuem Daten- und Programmspeicher auszustatten, angepasst an die neuen Berechungen. Ein Austausch von Eingabe/-Ausgabemodulen (*I/O Register*) ist allerdings nicht möglich, da die Anzahl und Bandbreite der Pins nicht verändert werden darf.
Die unterste Ebene bildet der Austausch von einzelnen Operatoren innerhalb der ALU (*ALU*). Dabei werden die Operatoren zur Designzeit klassifiziert in permanente Operatoren (*Permanent Operators*) und nicht permanente Operatoren. Permanente Operatoren sind dabei häufig benötigte Recheneinheiten, die sehr wahrscheinlich in jedem Algorithmus verwendet werden. Exemplarisch kann hier der Additionsoperator genannt werden. Nicht permanente Operatoren müssen weiterhin nach ihrem Ressourcenverbrauch eingeteilt werden. In Abbildung 4.13 wird exemplarisch lediglich in kleine und große Operatoren (*Small & Large Operators*) unterschieden. Eine detaillierte Unterscheidung ist prozessor- und realisierungsabhängig. Operatoren, die etwa vergleichbar viele Ressourcen benötigen, sind gut geeignet um gegeneinander ausgetauscht zu werden, sofern diese nicht zeitgleich in denselben Algorithmen zum Einsatz kommen.
Es ist weiterhin möglich, den Ressourcenverbrauch von Operatoren in einem gewissen Rahmen anzupassen, um eine bessere Angleichung mit anderen auszutauschenden Operatoren zu erreichen. Hierbei können Ressourcen über die Berechnungszeit des Operators oder dessen interne Pipelines angepasst werden. Es kann aber auch sinnvoll sein, einzelne Ressourcen, wie beispielsweise DSP-Blöcke, für den jeweiligen Operator zu verbieten. Das ist dann sinnvoll, wenn beispielsweise der durch diesen auszutauschenden Operator ebenfalls keine DSPs verwendet und so andere Areale auf dem physischen Chip verwendet werden können.

Ein weiterer wichtiger zu betrachtender Punkt ist das Abspeichern und Laden der Rekonfigurationsinformationen auf dem FPGA. Um die Rekonfigurationszeit minimal zu halten, muss sichergestellt werden, dass eine Speicheranbindung gewählt wird mit der es möglich ist, den Rekonfigurationsport auszulasten.

### Der partielle Rekonfigurationscontroller

Um eine partielle Rekonfiguration von Softcore Prozessoren zu ermöglichen, muss die Architektur entsprechend angepasst werden. Dazu müssen die entsprechenden zu rekonfigurierenden Teile des Softcore für eine partielle Rekonfiguration vorbereitet werden. Um welche Teile es sich dabei exakt handelt, hängt von der Rekonfigurationsgranularität, wie bereits vorgestellt, ab. Vorbereitet bedeutet vor allem, dass alle partiellen Rekonfigurationskomponenten (*PRCs*) die Voraussetzungen, wie bereits dargestellt, erfüllen, also vor allem, dass die Regionen der zugehörigen rekonfigurierbaren Blöcke so festgelegt werden, dass genügend Ressourcen vorhanden sind, aber auch das die jeweiligen Interfaces identisch sind.
Weiterhin wird für die eigenständige Rekonfiguration, also eine Rekonfiguration ohne einen externen Einfluss, zusätzliche Logik benötigt, die die Rekonfiguration vornimmt. Diese Logik wird als partieller Rekonfigurationscontroller (*Partial Reconfiguration Controller* (*PRCO*)) bezeichnet.
Der PRCO muss dabei folgende Eigenschaften aufweisen:

1. Der PRCO muss ein vorhersagbares und exaktes Timing besitzen, sodass sichergestellt werden kann, dass die harten Echtzeitschranken nicht verletzt werden.
2. Die für den PRCO verwendeten FPGA-Ressourcen sollten möglichst gering gehalten werden, da diese einen Overhead an Ressourcenverbrauch darstellen.
3. Der PRCO muss Bitstreams in einer Geschwindigkeit bereitstellen können, die es ermöglicht die partielle Rekonfiguration der jeweilig verwendeten Hardware mit höchstmöglicher Geschwindigkeit (abhängig des jeweilig verwendeten FPGAs) durchführen zu können.

Weitere Eigenschaften sind optional (je nach Anwendungsfall) möglich:

- Bitstream Kompression/Dekompression (z. B. bei sehr kleinen Speicherlösungen)
- Verwenden von Checksummen (z. B. bei Speicherlösungen ohne hinreichende Zuverlässigkeit)
- Bitstream-Design Kompatibilitätschecks (z. B. in Sicherheitsrelevanten Applikationen)

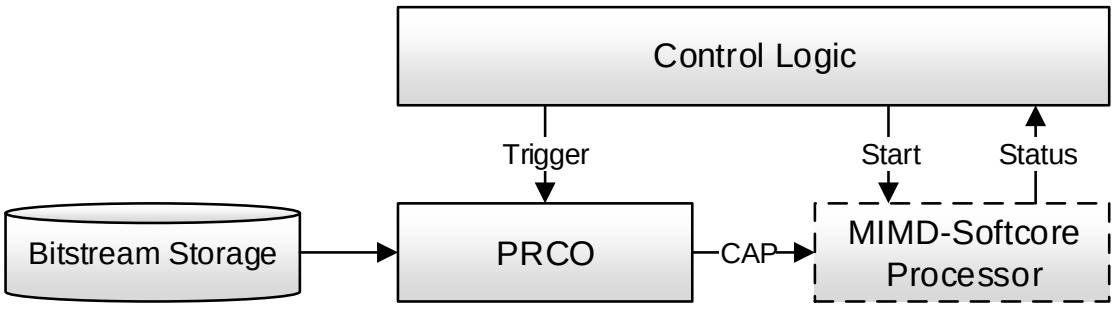



Abbildung 4.14: Prinzip der Rekonfiguration von Softcore Prozessoren

Wie in Abbildung 4.14 gezeigt ist, lädt der PRCO die jeweiligen Bitstreams der PRCs und gibt diese über den CAP (*Configuration Access Port* (*CAP*)) an die zu rekonfigurierende

Region weiter, sobald ein Trigger-Signal gesendet wird. Dieses Trigger-Signal wird von einer übergeordneten Kontrollogik (*Control Logic*) generiert, die ebenfalls dafür verantwortlich ist die Softcores zu starten und zu überwachen, um festzustellen zu welchem Zeitpunkt welche Rekonfiguration erfolgen muss.
In Abbildung 4.14 wird der MIMD-Softcore Prozessor (*MIMD-Softcore Processor*) als Einheit mit der größtmöglichen Rekonfigurationsgranularität gezeigt. Jedoch sind ebenfalls kleinere Rekonfigurationsgranularitäten oder sogar eine Kombination verschiedener Rekonfigurationsgranularitäten, je nach konkretem Anwendungsszenario, denkbar.
Die Hauptaufgabe des PRCO ist also das Laden und die Weitergabe der partiellen Bitströme der PRCs. Dabei gibt es verschiedene Möglichkeiten, wie diese Bitströme gespeichert werden können. Diese sollen im Folgenden diskutiert werden.
Grundlegend gibt es zwei wesentliche Möglichkeiten die Bitströme zu speichern: innerhalb oder außerhalb des FPGAs, bzw. etwas erweitert der Platine, auf der sich der FPGA befindet. Als interner Speicher steht der innerhalb des FPGA verwendbare BRAM zur Verfügung. Dieser ist allerdings in den meisten Fällen für diese Anwendung ungeeignet, da für diesen Anwendungsfall zu wenig BRAM auf einem FPGA vorhanden ist und dieser ebenfalls für die restliche auf dem FPGA befindliche Logik benötigt wird. Dies wird am Beispiel des Zynq Z-7020 deutlich: der FPGA besitzt 0,6125 MB BRAM. [Xil18f] Ein Bitstream, der diesen FPGA komplett beschreibt ist etwa 4,05 MB groß. Wird jetzt davon ausgegangen, dass 10 % des FPGAs als rekonfigurierbare Fläche verwendet werden sollen und soll diese mit insgesamt lediglich zwei unterschiedlichen PRCs verwendet werden, würden allein die beiden Bitstreams der zwei PRCs 0,81 MBs BRAM benötigen ($4,05 \text{ MB} \cdot 0,1 \cdot 2 = 0,81 \text{ MB}$), also bereits mehr als auf dem gesamten FPGA zur Verfügung steht. Das bedeutet, selbst bei einem vergleichsweise kleinen Beispiel wie diesem müssten die Bitstreams bereits in irgendeiner Weise komprimiert werden, damit diese überhaupt auf den internen BRAM gespeichert werden können. Da der BRAM aber zusätzlich für die eigentliche auf dem FPGA realisierten Module benötigt wird, kann davon ausgegangen werden, dass dieser Weg keine sinnvolle Option darstellt.
Eine externe Speicherlösung für die PRC Bitstreams ist abhängig von den jeweilig verwendeten Boards, auf denen sich die FPGAs befinden und was diese für Speicherlösungen anbieten. Das bedeutet, dass die exakte Speicherlösung Board-abhängig ist und aus diesem Grund das Interface vom PRCO zum Speicher generisch gehalten werden sollte, um eine möglichst große Kompatibilität zu gewährleisten.
Als Beispiele für externe Speicherlösungen sind an dieser Stelle Flash-Speicher, externer SDRAM oder embedded Multi Media Cards (*eMMCs*), aber auch Solid-State Disks (*SSDs*) zu nennen. Bei der jeweilig verwendeten Speicherlösung ist darauf zu achten, dass die von der Speicherlösung erzielten Datenraten mindestens den Datenraten entsprechen, mit der der interne Rekonfigurationsport (*Cap*) betrieben werden kann.

## 4.5 Methodische Assemblercodegenerierung aus Datenflussgraphen

Wie bereits in den vorhergehenden Kapiteln angedeutet, muss eine zeiteffiziente Entwicklung eines spezialisierten Softcore Prozessors und der dazu gehörigen Softcoresoftware auf verschiedenen Abstraktionsebenen stattfinden. In Abschnitt 4.3 wurde unter Abbildung 4.3 grob angerissen, wie eine mögliche effiziente modellorientierte Methode zur Generierung von Assemblercode aussehen kann.
Die Idee dahinter basiert auf der modellgetriebenen Softwareentwicklung (*Model-Driven Software*

*Development* (*MDSD*)) und soll das Softwaresystem als eine Kombination aus (Teil-) Modellen beschreiben, die eine syntaktische Repräsentation von Domänenwissen, Metamodellen und Transformationen sind. Die Metamodelle beschreiben dabei die Klasse der Modelle und die Transformationen definieren die Übersetzungen der Modelle in weitere Modelle oder Programmcode einer festgelegten Programmiersprache. [CH06, KBJV06]
Das bedeutet, dass MDSD von Softcore Prozessoren zur Softcoresoftwareentwicklung besteht aus:

- Modellen, die das Nutzerinterface beschreiben.
- Modellen für verschiedene Algorithmen oder Algorithmenteile, die frei kombiniert werden können.
- Modellen zur Schleifenkontrolle.

Einige dieser Modelle können vordefinierte Semantiken haben, oder aus anderen (Sub-) Modellen bestehen. Der Vorteil bei der MDSD liegt in einer Reduzierung des Risikos durch das bessere Verständnis von komplexen Problemen und der Möglichkeit, Teillösungen zu untersuchen bevor eine komplette Implementierung vorgenommen werden muss.
Weitere Vorteile der MDSD sind eine Abstraktion durch Ausblenden fachlich nicht relevanter Details, eine Verbesserung in Qualität der Darstellung sowie eine verbesserte Wartbarkeit. Entsprechende Systeme setzen eine einheitliche, getestete und dokumentierte Architektur um, die einen Quelltext von gleichbleibender Qualität generiert, Nutzer unabhängig. Die Flexibilität wird durch die Möglichkeit verbessert, verschiedene Schritte innerhalb des Generators zu automatisieren und diese auch ggf. automatisiert ändern zu können. [KT08, TPB+07]
Ein entsprechendes Tool sollte in einem in der Industrie weit verbreiteten Modellierungstool umgesetzt werden. Vorgeschlagen werden Matlab/Simulink oder LabVIEW. Denkbar wäre aber auch Scilab/Xcos.

Ein Nachteil bei großem Abstraktionslevel, der bei einer modellgetriebenen Entwicklung vorausgesetzt ist, ist die Erhöhung des Laufzeit-Overheads und die Reduktion der Performanz. Bei jeder automatisierten Übersetzung von einem hohen auf einen niedrigeren Abstraktionslevel entsteht ein Abstraktionsoverhead. Speziell in der vorgestellten Aufgabenklasse der echtzeitkritischen und kostenintensiven Applikationen ist das ein Problem, das gelöst bzw. minimiert werden muss.
Entsprechend werden im Weiteren Methoden vorgestellt, die eben diesen Abstraktionsoverhead bei der Übersetzung von Modellen in Assemblercode adressieren.

Um den entstehenden Abstraktionsoverhead zu minimieren, wird in Abbildung 4.15 vorgeschlagen Optimierungen an zwei voneinander unabhängigen Stellen vorzunehmen: auf Modellebene direkt am durch das Modell erzeugten Graphen (*Graph-based Optimizer*) und an dem daraus generierten Assemblercode (*Assembly Code*) über auf einem niedrigeren Abstraktionslevel arbeitende Optimierungsverfahren (*Code-based Optimizer*). Der Grund für die Aufteilung der Optimierungen auf zwei unterschiedliche Ebenen liegt in den Vorteilen und Möglichkeiten, die jede Abstraktionsebene hat. Zusätzlich dient die Optimierung auf dem generierten Assemblercode (*Assembly Code*) dazu, eine kompaktere Schnittstelle zu weiteren Verarbeitungsschritten zu ermöglichen. Je geringer der Speicherverbrauch des resultierenden Assemblercodes ist, desto besser kann man den dahinter stehenden Assembler bzw. dessen Features austauschen.
Auch in dem Modellierungsschritt von Abbildung 4.15 ist die Wichtigkeit einer möglichst

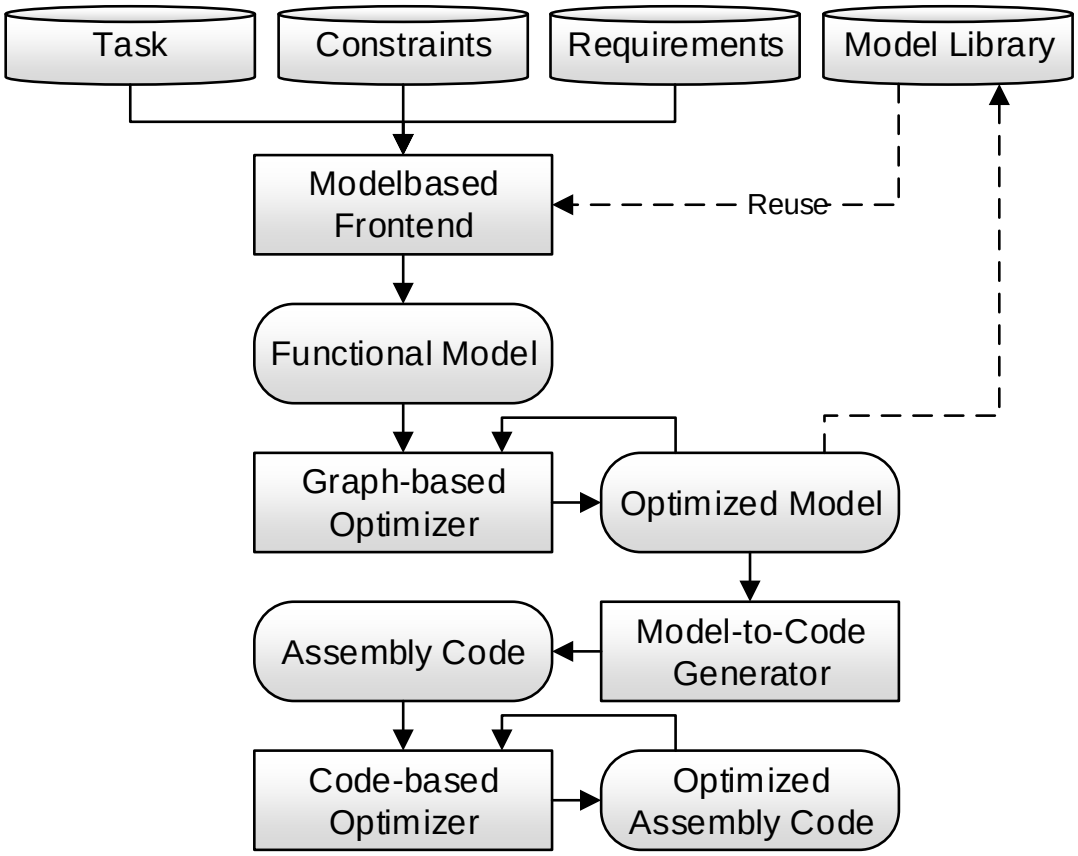



Abbildung 4.15: Modellbasierte Assemblercodegenerierung mittels Datenflussgraphen

effizienten Projektbearbeitung dargestellt. Aus diesem Grund können alle Schritte zur Assemblercodegenerierung automatisiert werden und die Wiederverwendbarkeit wird über eine spezielle (Teil-) Modell Bibliothek (*Model Library*) maximiert.
Die im Folgenden vorgestellten Optimierungsanätze wurden bereits unter [KWS+18] veröffentlicht.

### 4.5.1 Optimierungsansätze und Voraussetzungen

Nachdem der Nutzer ein funktionelles Modell (*Functional Model*, siehe Abbildung 4.15) in Form eines Datenflussgraphen erstellt hat, müssen an diesem verschiedene Umformungen ausgeführt werden, um die Optimierungsalgorithmen darauf ablaufen lassen zu können. Aufgrund der Anforderungen einer effizienten Codegenerierung und Wiederverwendungsmaximierung ist es möglich, bereits erstellte (Teil-) Modelle in das neue Projekt einzubinden. Diese Modelle können rekursiv weitere (Teil-) Modelle aus früheren Iterationsrunden beinhalten. Der erste Schritt ist, die rekursiven (Teil-) Modelle (*SubSystem*) durch äquivalente Funktionsblöcke auszutauschen und diese entsprechend der Spezifikation mit der restlichen Modellogik zu verbinden. Dieser Schritt wird als Verflachung des Modells bezeichnet. Das wird exemplarisch in Abbildung 4.16 an einem einfachen Beispiel illustriert.

Es wird dabei von der Annahme ausgegangen, dass jeder Datenflussgraph aus mindestens einer Datenquelle, einem Verbinder (also einem Datenfluss) und einer Datensenke besteht. Als Quelle wird im Modell der Input externer Werte oder die Definition interner Variablen und Konstanten angenommen. Senken sind Ausgabeschnittstellen an übergeordnete (Teil-) Modelle oder die Übergabe eines Wertes an externe Logik.
Die Blöcke des Modells repräsentieren Operationen. Diese Operationen können zwei Argumente benötigen und für einige dieser Operationen gilt das Kommutativgesetz nicht. Aus diesem Grund ist das Modell so eingefärbt, dass das erste Argument einer Operation einen schwarzen Verbinder und das zweite Argument der gleichen Operation einen roten Verbinder besitzt. Eine Markierung über spezielle Symbole an den jeweiligen Eingängen der Blöcke wäre ebenfalls denkbar, wurde hier aber nicht verwendet.

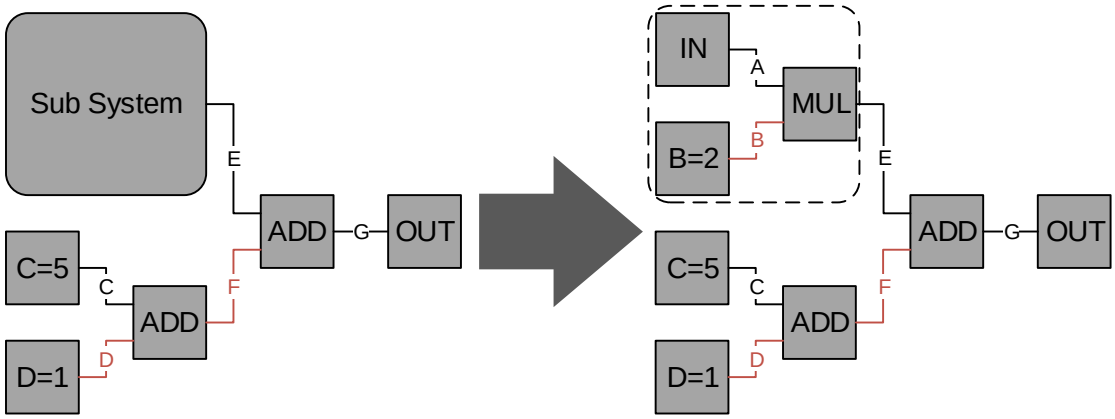



Abbildung 4.16: Teilmodellersetztung durch äquivalente Funktionsblöcke

Die hier dargestellten Blöcke entsprechen Operationen, die im Assemblercode existieren. So steht Beispielsweise *C=5* für die Definition einer Konstante mit dem Wert 5, *ADD* steht für eine Addition, *MUL* für eine Multiplikation und *OUT* steht für die Ausgabe des Wertes.
Der nächste notwendige Umformungsschritt leitet sich aus der Anforderung der harten Echtzeitfähigkeit ab. Um die Einhaltung harter Echtzeitschranken garantieren zu können, muss der in einem Prozessor verwendete Maschinencode, und damit auch der eine Abstraktionsebene darüber befindliche Assemblercode, zur Designzeit voll analysierbar und zeitlich verifizierbar sein.
Aus diesem Grund ist die Verwendung von Sprüngen auf zur Designzeit nicht bekannte Stellen im Assemblercode verboten, da diese eine Analyse unmöglich machen würden.
Um aber eine möglichst effiziente modellbasierte Implementierung von Assemblercode dennoch zu ermöglichen, werden spezielle Schleifenkonstrukte vorgeschlagen, die eine definierte Anzahl von Iterationen besitzen und voll zur Designzeit analysierbar sind. Um einen entwickelten Datenflussgraphen in einen optimierbaren Graphen umzuwandeln, ist es notwendig, diese Schleifen auszurollen und entsprechend der Spezifikation mit den restlichen Funktionsblöcken zu verknüpfen. In Abbildung 4.17 wird das an einem einfachen Beispiel verdeutlicht. Hier zu sehen ist, dass automatisch auch eine Anpassung der umgebenden Funktionsblöcke und Verbindungen vorgenommen wird, die direkt mit dem Schleifenmodell in Verbindung steht. Es werden in diesem Beispiel weitere Input-Blöcke (*IN*) erzeugt. Diese werden intern wiederum auf dieselbe Adresse wie ihre originale gelegt. Es werden also die gleichen Input-Ports verwendet, da allerdings eine zeitliche Differenz besteht, kann es, je nach Algorithmus, vorkommen, das andere Werte an den

jeweiligen Ports anliegen. Das Gleiche gilt für den zusätzlich erzeugten Ausgabe-Block ($OUT$). Dabei ist es möglich, dass die Ergebnisse von einem Schleifendurchlauf als Eingaben von weiteren Schleifendurchläufen rekursiv verwendet werden können.

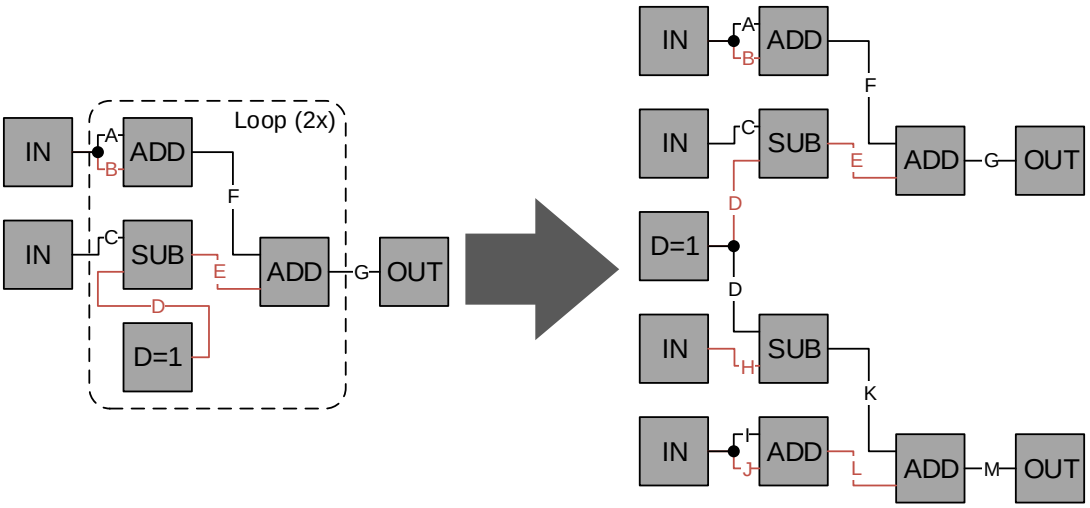



Abbildung 4.17: Schleifenausrollen

Nach diesen internen Umformungen kann das Modell automatisiert zu einem Graphen transformiert werden, um auf der Graphenebene Optimierungen vorzunehmen. So ist es möglich, auf dieser Ebene verschiedene durch den modellierenden Nutzer des Datenflussgraphtools entstandene Schwachstellen im Modell zu identifizieren und zu beheben, was auf Assemblercodeebene nicht oder nur sehr schwer möglich ist.
Dazu wird ein gefärbter Multi-Graph [Die17] $G$ definiert, der aus einer endlichen Menge an Knoten ($V$) und einer endlichen Menge an Verbindern ($E$) besteht. Jeder Verbinder hat eine Quelle (*Source* ($src$)) und ein Ziel (*Target* ($trg$)). Zusätzlich hat der Graph eine Funktion ($f$), die jedem Knoten eine Operation, äquivalent zu den repräsentierenden Blöcken des Datenflussgraphen zuordnet. Eine weitere im Multi-Graphen verwendete Abbildung ($dir$) bildet jede Kante auf eine Menge an Farben derart ab, dass Kanten mit gleicher Quelle und Ziel verschieden gefärbt sind:

$$G = (V, E, src, trg, f, dir) \tag{4.2}$$

Die Verwendung eines Multi-Graphen ist deswegen notwendig, da es mehrere Verbinder zwischen den gleichen Quell- und Zielknoten geben kann.
Schleifen sind innerhalb des Multi-Graphen verboten.

$$V \cap E = \emptyset \tag{4.3}$$

Jeder Knoten $v \in V$ repräsentiert dabei einen Block aus dem zugrunde liegenden Modell. Jeder Verbinder $e \in E$, im Folgenden auch als Kante bezeichnet, repräsentiert eine Ergebnisspeicherzelle. Für jeden Verbinder $e \in E$ gibt es dabei die Zuordnung zu einem Quellknoten ($src$) und einem Zielknoten ($trg$):

$$\begin{aligned} src &: V \to E \\ trg &: V \to E \end{aligned} \tag{4.4}$$

Es kann also gesagt werden, dass ein Verbinder $e$ mit $src(e) = u$ und $trg(e) = v$ existiert, genau dann wenn es im zugrunde liegenden Datenflussgraph eine direkte Verbindung (oder auch: einen direkten Datenfluss) von Knoten $u$ nach Knoten $v$ gibt.
Die Funktion $f$ ist dabei eine Funktionsabbildung jedes Knotens $V$ in die Menge, die alle verfügbaren Assemblerbefehle beinhaltet. Dabei wird zwischen Befehlen unterschieden, die sowohl eingehende, als auch ausgehende Kanten haben (Menge *cmd*) und der Menge der Befehle, die diese Eigenschaft nicht erfüllen:

$$f : V \to cmd \cup \{IN, Const, OUT\} \tag{4.5}$$

Die Stelligkeit (Arität) *ar* von *cmd* ist definiert als eine Abbildung von der Menge der Assemblerbefehle in die Menge der positiven natürlichen Zahlen und stellt die Anzahl an eingehenden Kanten, also an Operationsargumenten, dar:

$$ar : cmd \to \mathbb{N}_{>0} \tag{4.6}$$

Die Funktion (*dir*) ist eine Abbildung der Kantenmenge in die Menge der Farben (*Color C*). Diese Abbildung dient der eineindeutigen Unterscheidung von Kanten mit gleichem Ziel (und potenziell sogar gleicher Quelle). Diese Unterscheidung ist für nicht kommutative Operationen, die durch etwaige Zielknoten repräsentiert werden können, notwendig.

$$dir : E \to C \tag{4.7}$$

Damit können folgende Festlegungen getroffen werden:

$$f(v) \notin \{IN, Const, OUT\} \Rightarrow v \text{ hat } ar(f(v)) \text{ Eingangskanten } \& \geq 1 \text{ Ausgangskante} \tag{4.8}$$

$$f(v) \in \{IN, Const\} \Rightarrow v \text{ hat } 0 \text{ Eingangskanten } \& \geq 1 \text{ Ausgangskante} \tag{4.9}$$

$$f(v) = OUT \Rightarrow v \text{ hat } 1 \text{ Eingangskante } \& 0 \text{ Ausgangskanten} \tag{4.10}$$

Es sei angemerkt, dass der maximale Wert von *ar* von der jeweilig verwendeten Prozessorarchitektur abhängig ist. Das bedeutet, wie viele Argumente eine Operation verwenden darf, kann variieren. Es wird allerdings empfohlen, keine komplexen Operationen mit mehr als zwei Argumenten zu verwenden und entsprechende Konstrukte stattdessen in mehrere elementare Operationen aufzuteilen und beispielsweise komplexe Operationen auf Ebene des Datenflussgraphen als Subsysteme darzustellen. Der Grund ist die im Hintergrund

notwendige Geschwindigkeit der Speicherarchitektur. Sollen es möglich sein in jedem Takt eine neue Operation mit bis zu zwei Argumenten zu starten, so muss der verwendete Speicher schon doppelt so schnell sein, wie die vom Prozessor verwendete Frequenz. Entsprechend würde sich dieses Geschwindigkeitsverhältnis bei drei Argumenten schon zu drei zu eins verändern. Mit Operationen mit vielen Argumenten wird also die effektiv mögliche maximale Taktrate der verwendeten Prozessoren beschränkt.

Sei $W \subseteq V$ die Menge der Knoten $w \in V$ mit

$$\nexists\, v \in V : e_{vw} \in E,\, src(e_{vw}) = v \,\wedge\, trg(e_{vw}) = w \tag{4.11}$$

$W$ ist damit wie folgt definiert:

$$W := \{w \in V \,|\, \forall v \in V : \neg e_{vw} \in E \text{ mit } src(e_{vw}) = v \wedge trg(e_{vw}) = w\} \tag{4.12}$$

Nach der Annahme, dass jeder zugrunde liegende Datenflussgraph wenigstens eine Verbindung enthält, ist $W$ eine echte Teilmenge von $V$ ($W \subset V$).
Alle Elemente aus der Menge $W$ sind also entweder Konstantendeklarationsknoten oder Eingabeknoten.
Weiterhin sei $Z \subseteq V$ die Menge der Knoten $z \in V$ mit

$$\nexists\, v \in V : e_{zv} \in E,\, src(e_{zv}) = z \,\wedge\, trg(e_{zv}) = v \tag{4.13}$$

$Z$ ist damit wie folgt definiert:

$$Z := \{z \in V \,|\, \forall v \in V : \neg e_{zv} \in E \text{ mit } src(e_{zv}) = z \wedge trg(e_{zv}) = v\} \tag{4.14}$$

Für die Menge $Z$ gilt ebenfalls aufgrund der Annahme, dass jeder zugrunde liegende Datenflussgraph wenigstens eine Verbindung enthält, dass $Z$ eine echte Teilmenge von $V$ ist ($Z \subset V$).
Alle Elemente, die sich in der Menge $Z$ befinden, besitzen exakt eine eingehende Kante und keine ausgehenden Kanten und sind damit Ausgabeknoten (oder auch Senkknoten genannt).

Anzumerken ist, dass die in den folgenden Abbildungen zu sehenden Graphen aus Gründen der Übersichtlichkeit vereinfacht dargestellt sind. Zum besseren Verständnis soll im folgenden die Bedeutung hinter den Graphen mit Abbildung 4.18 erklärt werden.

Zu sehen ist links der eigentliche Multi-Graph und rechts die vereinfachte Darstellung. Der wesentliche Unterschied liegt darin, dass die Kanten (also die Verbinder) nicht wie bei normalen Graphen Tupel von Knoten sind (z. B. $A = (Constant, Multiplication)$), sondern eigenständige Objekte mit Quell- und Zielfunktionen ($src$ und $trg$).

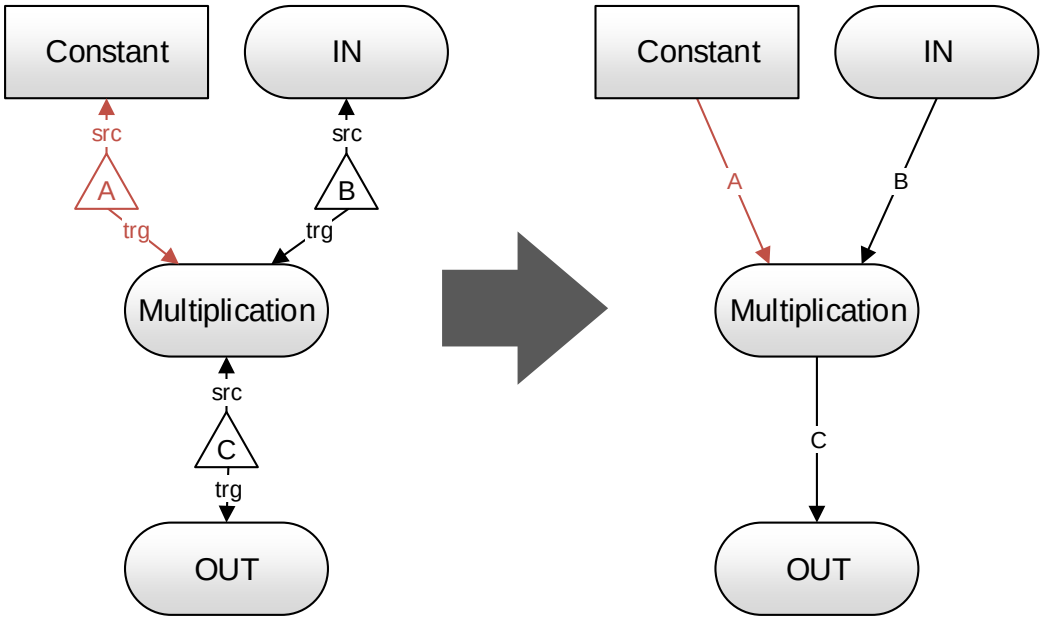



Abbildung 4.18: Multi-Graph vereinfachte Darstellung

### 4.5.2 Graphbasierte Optimierung

Zur Transformation des ausgerollten und verflachten Modells, also des Modells ohne Schleifen- und Teilmodellblöcke, zu dem korrespondierenden Graphen wird das Beispiel aus Abbildung 4.16 in der Abbildung 4.19 weiterverwendet.

Zu sehen ist die Transformation des verflachten und ausgerollten Modells zu der logisch gleichen Graphenrepräsentation im ersten Schritt. Die Transformation ist dabei, wie unter Abbildung 4.18 erklärt, vereinfacht dargestellt. Im Anschluss daran findet eine Optimierung auf dem Graphen statt, die „unnötig komplexe“ Konstrukte umwandeln soll. Unter solchen Konstrukten wird hier eine Kombination aus Funktionsblöcken verstanden, die zu einer unnötigen Verlängerung der Laufzeit des resultierenden Assemblercodes führen würde. Solche Konstrukte können entstehen, wenn der Nutzer innerhalb des Modells aufwendige Kombinationen erstellt, die über andere Funktionsblöcke zum semantisch gleichen Ergebnis führen. Darunter wird vor allem die Kombination und Wiederverwendung von zuvor bereits erstellten Teilmodellen gezählt, aufgrund dessen es zu solchen komplexen Konstrukten kommen kann.
In Abbildung 4.19 wird exemplarisch die Berechnung von Operationen dargestellt, die nur konstante Werte als Eingabe verwenden, hier die Addition (*ADD*) von den Konstanten (*C=5*) und (*D=1*). Der Optimierer wird solche Konstrukte finden und gegen semantisch gleiche Funktionsblöcke ersetzen, in diesem Fall wird die gesamte Operation und die Deklaration der Konstanten durch die Deklaration des Ergebnisses als eigene Konstante (*Constant3 (Constant)*) ersetzt. Diese Optimierung wird dann rekursiv auf dem resultierenden Graphen erneut ausgeführt, da mit einer semantischen Ersetzung weitere ersetzbare Konstrukte entstehen können.
Es ist Anzumerken, dass hier ein Beispiel dargestellt wird, mit Hilfe dessen man beliebige

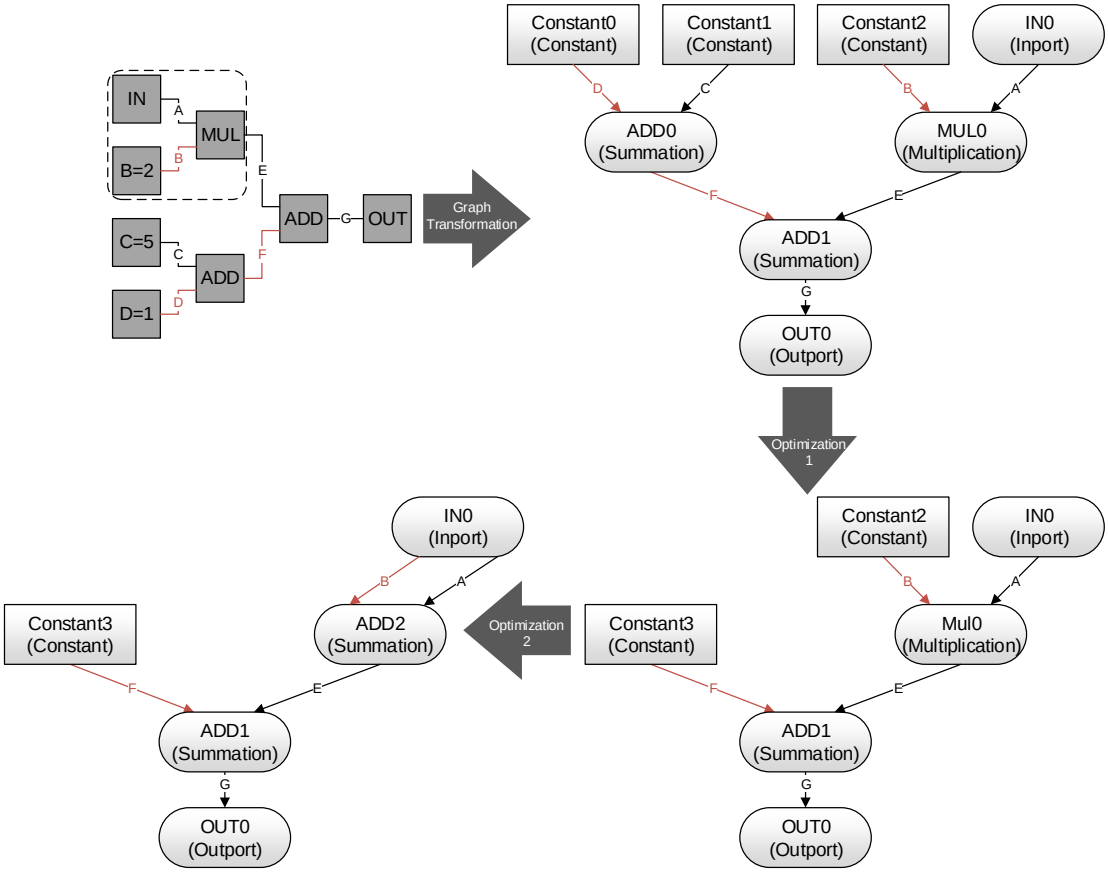



Abbildung 4.19: Optimierungen auf Graphenebene

Optimierungen umsetzen kann. Diese Optimierungen können an dieser Stelle allerdings nicht im Speziellen diskutiert werden, da zur Erstellung konkreter Optimierungsausdrücke detaillierte Informationen über Implementierungsdetails, wie beispielsweise Operatoraufbau und Operatorrechenzeit, bekannt sein müssen. Die gezeigte Optimierung dient also als Anleitung eines schrittweisen Aufbaus zur Definition und Umsetzung von Optimierungsausdrücken.
Das Optimierungsbeispiel wird im Folgenden formal definiert. Es wird dabei angenommen, dass zwei Multi-Graphen $D_1$ und $D_2$ existieren:

$$\begin{aligned} D_1 &= (V_1, E_1, src_1, trg_1, f_1, dir_1) \\ D_2 &= (V_2, E_2, src_2, trg_2, f_2, dir_2) \end{aligned} \tag{4.15}$$

Im Anschluss wird definiert, unter welchen Bedingungen $D_2$ aus $D_1$ durch exakt einen Optimierungsschritt hervorgeht ($D_1 \vdash D_2$). Dazu müssen die folgenden Bedingungen erfüllt sein:

Es gibt $e_1, e_2 \in E$ $(e_1 \neq e_2)$, mit:

$$\begin{aligned} f_1(src_1(e_1)) &= Const_1 \\ f_1(src_1(e_2)) &= Const_2 \\ trg_1(e_1) &= trg_1(e_2) \\ f_1(trg(e_1)) &\in cmd \end{aligned} \tag{4.16}$$

Dann ergibt sich $D_2$ wie folgt:

$$E_2 = E_1 \setminus \{e_1, e_2\} \tag{4.17}$$

$$V_2 = V_1 \setminus \{x \in V_1\,,\, f_1(x) \neq OUT \wedge x \text{ hat keinen } E_2\text{-Nachfolger.}\} \tag{4.18}$$

$$src_2 = src_1 \restriction E_2 \tag{4.19}$$

$$trg_2 = trg_1 \restriction E_2 \tag{4.20}$$

$$f_2(x) = \begin{cases} Const_1\ f_1(trg(e_1))\ Const_2, \text{ falls } x = trg(e_1), \\ f_1(x), \text{ sonst.} \end{cases} \qquad \forall x \in V_2 \tag{4.21}$$

$$dir_2 = dir_1 \restriction E_2 \tag{4.22}$$

Falls also eine Operation nur konstante Eingabewerte verwendet, kann das Ergebnis dieser bereits zur Designzeit berechnet und entsprechend ersetzt werden ($Const_1\ f_1(trg(e_1))\ Const_2$, falls $x = trg(e_1)$). Sollte also exemplarisch $f_1(trg(e_1)) = ADD$ sein, dann würde sich daraus ergeben, dass $Const_1 + Const_2$, falls $x = trg(e_1)$. Dabei werden potenziell alle Konstantenknoten gelöscht, sofern diese keine weiteren wegführenden Verbinder besitzen. Außerdem werden alle Verbinder zwischen den alten Konstantenknoten und der ersetzten Berechnung gelöscht. Mit dieser Definition lassen sich nun beliebig viele Optimierungsschritte durchführen ($D \vdash^* D'$), bis der Graph mit diesem Verfahren nicht weiter optimierbar ist.
Anzumerken ist, das diese Optimierungsdefinition für die Ersetzung von Knoten und Verbindern im Graph gilt, wenn die dazugehöhrigen Operationen exakt zwei Argumente benötigen. Analog lassen sich diese Regeln für beliebige Operationen mit $1..n$ Eingaben aufstellen. Da diese Definitionen allerdings größtenteils redundant zu der hier gezeigten Definition für zwei Argumente sind, wird darauf verzichtet.

In einem weiteren Optimierungsschritt würde der Algorithmus die Multiplikation einer externen Variable, mit zur Designzeit nicht definiertem Wert, mit der Konstanten *B=2* erkennen und ebenfalls ersetzen. Hier würde die Multiplikation durch eine Addition (*ADD2 (Summation)*)

ersetzt werden, da eine Addition einer Variable mit sich selbst schneller zu berechnen ist, als eine Multiplikation der Variable mit dem Wert *2*. Zusätzlich wird der Speicherplatz der Konstante *B=2* nicht mehr benötigt, was die Speicherauslastung verringert.
Auch an dieser Stelle gilt: das ist ein Beispiel, das nur bei einem vorgegebenen Set an Implementierungsdetails sinnvoll ist und soll die korrekte Ersetzung einer Operation durch eine andere Operation verdeutlichen, unter der Voraussetzung, dass der semantische Sinn des gesamten Algorithmus nicht verändert wird. Anhand dieses Beispiels kann der Nutzer, mit Hilfe detaillierter Implementierungsdetails eigene Optimierungen definieren und umsetzen.
Formal wird dieses Verfahren nach dem gleichen Prinzip definiert, wie bereits das vorherige Optimierungsverfahren. Exemplarisch erfolgt diese Definition für das in Abbildung 4.19 gezeigte Beispiel:
Es existieren zwei Multi-Graphen, wie schon in Definition 4.15 gezeigt. Weiterhin wird angenommen, dass zwei Verbinder existieren:

$$e_1, e_2 \in E\,(e_1 \neq e_2) \text{ mit:} \tag{4.23}$$

$$\begin{aligned} f_1(src_1(e_1)) &= 2 \\ f_1(src_1(e_2)) &\in cmd \cup \{IN\} \\ trg_1(e_1) &= trg_1(e_2) \\ f_1(trg(e_1)) &= MUL \end{aligned} \tag{4.24}$$

Dann ergibt sich $D_2$ wie folgt:

$$\begin{aligned} V_2 &= V_1 \\ E_2 &= E_1 \\ trg_2 &= trg_1 \end{aligned} \tag{4.25}$$

$$src_2(x) = \begin{cases} src_1(e_2), \text{ falls } x = e_1 \\ src_1(x), \text{ sonst.} \end{cases} \qquad \forall x \in E_2 \tag{4.26}$$

$$f_2(x) = \begin{cases} ADD, \text{ falls } x = trg(e_1) \\ f_1(x), \text{ sonst.} \end{cases} \qquad \forall x \in V_2 \tag{4.27}$$

$$dir_2 = dir_1 \restriction E_2 \tag{4.28}$$

Die genaue Festlegung, welche logischen Konstrukte zu ersetzen sind und welche nicht, kann nur mit Hilfe von zusätzlichen Metainformationen getroffen werden. So muss der Algorithmus die genauen Verarbeitungszeiten von Operationen wie Addition oder Multiplikation kennen um zu entscheiden, ob diese gegen funktional gleiche Äquivalente voneinander getauscht werden können.

Als letzter Optimierungsansatz wird das in Abbildung 4.19 nicht dargestellte Verfahren zur Erkennung und Vereinfachung von redundanten (Teil-)Graphen vorgestellt. Hierbei wird davon ausgegangen, dass wenn mehrere Knoten die gleichen Argumente verwenden und die interne Funktion der Knoten identisch ist, das diese redundant sind und entsprechend vereinfacht werden können, wie die folgende Abbildung 4.20 zeigt. Anhand dieses Beispiels werden ebenfalls nicht-kommutative Operationen betrachtet und wie diese mit Hilfe der *dir*-Abbildung identifiziert werden.

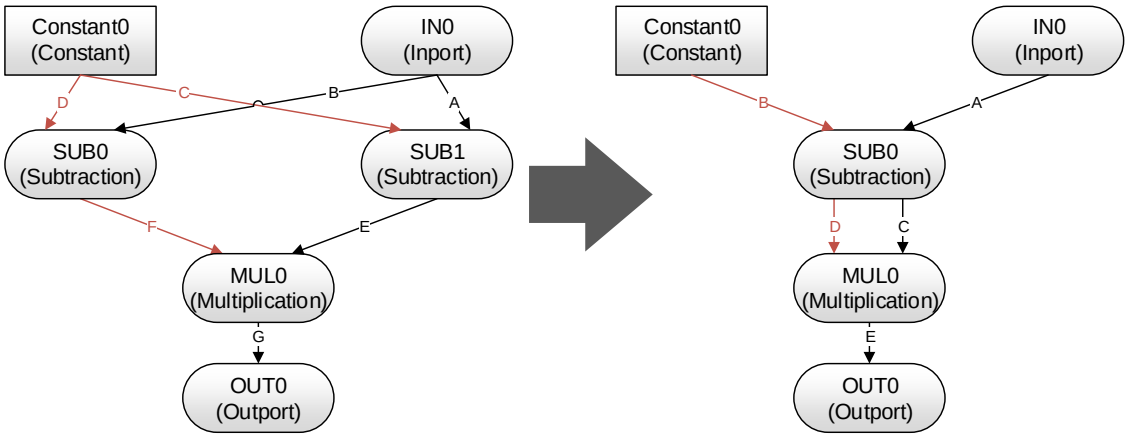



Abbildung 4.20: Optimierungen von Redundanzen auf Graphenebene

Formal betrachtet wird auch hier davon ausgegangen, dass zwei Multi-Graphen existieren:

$$\begin{aligned} D_1 &= (V_1, E_1, src_1, trg_1, f_1, dir_1) \\ D_2 &= (V_2, E_2, src_2, trg_2, f_2, dir_2) \end{aligned} \tag{4.29}$$

Jetzt gilt $D_1 \vdash D_2$, wenn zwei Knoten existieren $v_1, v_2 \in V$ mit:

$$f_1(v_1) = f_1(v_2) \tag{4.30}$$

Weiterhin müssen vier Verbinder $e_1, e_2, e_3, e_4 \in E$ existieren mit:

$$\begin{aligned} src(e_1) &= src(e_2) \\ src(e_3) &= src(e_4) \\ trg(e_1) &= v_1 \\ trg(e_2) &= v_2 \\ trg(e_3) &= v_1 \\ trg(e_4) &= v_2 \\ dir(e_1) &= dir(e_2) \\ dir(e_3) &= dir(e_4) \end{aligned} \tag{4.31}$$

In diesem Fall ergibt sich $D_2$ wie folgt:

$$V_2 = V_1 \setminus v_2 \tag{4.32}$$

$$E_2 = E_1 \setminus \{e_2, e_4\} \tag{4.33}$$

$$src_2 = src_1 \upharpoonright E_2 \tag{4.34}$$

$$trg_2 = trg_1 \upharpoonright E_2 \tag{4.35}$$

$$f_2 = f_1 \upharpoonright V_2 \tag{4.36}$$

$$dir_2 = dir_1 \upharpoonright E_2 \tag{4.37}$$

Wie anhand dieses Beispiels gezeigt, ist bei nicht kommutativen Operationen darauf zu achten, dass die Einfärbung der jeweiligen Kanten und damit die *dir*-Funktion identisch sein muss. Das bedeutet, wenn der Zielknoten eine nicht kommutative Operation repräsentiert, in diesem Beispiel eine Subtraktion, dann muss darauf geachtet werden, dass die jeweiligen Kanten identisch zugeordnet sind. Das bedeutet, dass bei jeder Subtraktion die zusammengefasst werden soll, die Kanten des Minuenden und des Subtrahenden jeweils aus demselben Quellknoten stammen und dieselbe Farbe besitzen.

### 4.5.3 Modell zu Code Transformation

Im Anschluss an die graphbasierten Optimierungen kann über verschiedene Sequenzialisierungsstrategien eine Optimierung des resultierenden Assemblercodes erreicht werden. Das wichtigste Kriterium dabei ist, dass der resultierende sequenzielle Assemblercode die gleiche Funktion erfüllt, wie das Modell. Ergebnis und funktionelle Definition des modellierten Algorithmus dürfen nicht verändert werden.

Dazu muss zuerst definiert werden, was eine Sequenzialisierung ist:
Sei $(V, E, src, trg, f, dir)$ ein Datenfluss-Multi-Graph. Eine Sequenzialisierung ist eine lineare Ordnung $\sqsubseteq$ auf $V$, so dass gilt: der Quellknoten eines Verbinders liegt vor dem Zielknoten eines Verbinders:

$$e \in E \Rightarrow src(e) \sqsubseteq trg(e). \tag{4.38}$$

Hierzu muss zuerst bestimmt werden, welche Knoten zu einem bestimmten Zeitpunkt wählbar sind und welche nicht. Dazu wird ein virtueller Startknoten $s$ erzeugt, von dem jeder Knoten $w_1..w_n \in W$ der keine hinführenden Kanten $e$ besitzt, direkt abgeleitet wird. Das bedeutet, dass zwischen jedem Knoten $w$ und dem Startknoten $s$ eine virtuelle Kante $e$ mit $src(e) = s$ und $trg(e) = w$ generiert wird:

$$V^+ = V \cup \{s\} \tag{4.39}$$

$$E^+ = E \cup \{e_w : w \in W,\ f(w) \in \{IN\}\} \tag{4.40}$$

$$\begin{aligned} src^+(e_w) &= s \\ trg^+(e_w) &= w \\ f^+ &= f \\ dir^+ &= dir \end{aligned} \tag{4.41}$$

Es wird weiterhin angenommen, dass jeder Knoten $v$ von allen Vorgängerknoten $u$ abhängig ist, wenn eine Kante $e_{uv}$ existiert mit $src(e_{uv}) = u$ und $trg(e_{uv}) = v$ und der Knoten $u$ noch nicht ausgewählt wurde. Damit kann eine Liste erzeugt werden, in der alle wählbaren, also unabhängigen, Knoten eingetragen werden, beginnend mit dem Startknoten $s$. In jeder Runde wird ein Knoten aus der Liste ausgewählt und die repräsentative Assemblerlogik des Knotens wird dem Assemblercode hinzugefügt. Dazu zählt auch die Deklaration von Variablen und Konstanten. Sobald ein Knoten ausgewählt wurde, wird dieser aus der Liste entfernt und die Liste wird um alle dadurch unabhängig gewordenen Knoten erweitert. Die exemplarische Sequenzialisierung zu dem in Abbildung 4.19 gezeigten Beispiel befindet sich in Anhang A unter Abbildung A.1.
Die Optimierung wird über die Auswahlstrategie der Knoten aus der Liste bestimmt. Hier gibt es verschiedene denkbare Ansätze, je nach detailliertem Aufbau des Assemblercodes und benötigter Rechenzeit der einzelnen Operationen. Es werden dabei aus der Literatur bekannte sowie eine speziell auf die Charakteristiken angepasste Sequenzialisierungsstrategie vorgeschlagen:

- **FiFo**
  Diese Strategie folgt keiner speziellen Assemblereigenschaft. Hier wird immer der erste Knoten aus der Liste der wählbaren Knoten gewählt. [Pin08]
- **Prioritätsbasiert**
  Bei der Strategie eines prioritätsbasierten Sequenzialisierung wird die Liste nach der Priorität der Knoten in Bezug auf die Verarbeitungzeit sortiert. So stehen die Operationen mit

langen Verarbeitungszeiten auf den ersten Positionen in der Liste und werden damit früher ausgewählt. Die Idee dahinter ist, dass Operationen $O_l$ mit langen Verarbeitungszeiten so früher im Assemblercode stehen und damit die Abarbeitung im Maschinencode früher beginnt und damit auch früher beendet wird. So können etwaige Operationen $O_d$, die von den Ergebnissen dieser Operationen $O_l$ abhängen, ebenfalls potenziell früher berechnet werden. Voraussetzung für diese Optimierungsstrategie sind die Informationen über die Verarbeitungszeiten der jeweiligen Blöcke, bzw. deren Assemblerbefehlrepräsentationen. [Pin08]

- **Rollout Priority**
  Mit der Rollout Priority Sequenzialisierungsstrategie wird eine Strategie vorgeschlagen, die eine besonders effiziente Sequenzialisierung für die weitere Optimierung der Variablenreduktion, vorgestellt in Abschnitt 4.5.4, erstellt.
  Die Rollout Priority Strategie geht von der Annahme aus, dass der Nutzer logisch zusammenhängende Blöcke und Teilmodelle, auch innerhalb des Modells räumlich nah aneinander anordnet. Dieser Annahme folgend, wird bei dieser Strategie über die Lage der Blöcke im ursprünglichen Modell versucht, die Anordnung im resultierenden Code so auszurollen, dass diese Blöcke möglichst nah beieinander platziert werden.
  Der Vorteil liegt in der Wiederverwendbarkeit der Variablen, da in den meisten Modellen multiple Iterationen von Befehlsabfolgen existieren, wobei die untereinander verwendeten Variablen wiederverwendet werden können, was die Auslastung der Speicherzellen verringert. Dadurch ist es möglich, dass der resultierende Assemblercode auch von Softcores verwendet werden kann, die nur einen sehr kleinen Speicher besitzen.

Weitere optimierende Sequenzialisierungsstrategien sind recherchierbar; abhängig von den speziellen Eigenschaften des jeweiligen Softcore Prozessors, bzw. dem Assemblercode, den dieser verwendet.

### 4.5.4 Variablenreduktion

Die graphbasierten Optimierungen zielen auf eine Optimierung der Struktur des Graphen, und damit auf eine Minimierung der Codelänge des resultierenden Assemblercodes ab. Allerdings werden Speicherbedarf und Abhängigkeiten der verwendeten Variablen untereinander dabei nicht einbezogen. Aus diesem Grund ist es notwendig, ein Verfahren zur Variablenreduktion und damit zur Speicherminimierung zu realisieren. Dieses Verfahren ist essentiell zur Optimierung des Speicherbedarfs des Prozessors und greift damit eines der wesentlichen Negativpunkte der MDSD an:
Aus dem Abstraktionsgrad entsteht ein deutlich höherer Bedarf an Speicher, da Variablen- und Konstantenspeicher nicht optimal genutzt werden. Die Variablenreduktion soll genau dieses Problem durch gezielte Optimierung minimieren.
Dieser Optimierungsschritt dient dabei nicht der globalen Optimierung des Gesamtergebnisses, sofern ein nachfolgender Assembler ebenfalls Mechanismen zur effizienten Speichernutzung verwendet. Dieser Optimierungsschritt dient in entsprechenden Fällen als Formatierung der Schnittstelle. Das Ziel dieser Optimierung ist ein Assemblercode, mit einer minimierten Anzahl an Variablen, der von beliebigen Assemblern übersetzt werden kann. Ein Assemblercode, der ohne dieses oder ein vergleichbares Verfahren mittels eines modellbasierten Ansatzes erzeugt wurde, kann nicht von Assemblern ohne eigene Speicheroptimierungen übersetzt werden. Aufgrund der allgemeinen Verwendbarkeit des Ergebnisses dieses Datenflussgraphen ist dieser

Schritt daher notwendig.

Es ist anzumerken, dass hier die Anzahl an semantischen Variablen im Assemblercode und nicht die im Softcore verwendeten physischen Speicherzellen reduziert werden, da auf dieser Abstraktionsebene keine Informationen über Aufbau und Anzahl an physischen Speicherzellen zur Verfügung steht.
Die Variablenreduktion basiert auf der Annahme, dass es die Architektur des verwendeten Softcore Prozessors erlaubt, das Ergebnis jeder Operation in eines der verwendeten Eingangsregister zu speichern. Sollte die Architektur das nicht erlauben, müssen an dem im Folgenden vorgestellten Algorithmus Anpassungen vorgenommen werden, wie im Weiteren gesehen werden kann.
Bei der Variablenreduktion wird der Assemblercode auf eine $m \times n$ Matrix abgebildet, wobei $m$ die Anzahl der Codezeilen und $n$ die Anzahl der Variablen darstellt. Um eine bestmögliche Optimierung zu gewährleisten, wurde bei der Modell-zu-Code Transformation angenommen, dass jeder ausgehenden Kante $e = (u, v)$ eine neue Variable im erzeugten Assemblercode zugeordnet wird. Innerhalb der Matrix wird jeder Variable ein Wert zugeordnet, abhängig vom Zustand der Variable in der Codezeile, wie in Tabelle 4.2 dargestellt.

| Wert | Belegung |
|---|---|
| 0 | Variable nicht vorhanden |
| 1 | Variable als Operand 1 gelesen |
| 2 | Variable als Operand 2 gelesen |
| 3 | Variable als Operand 1 und 2 gelesen |
| 4 | Variable wird geschrieben |
| 5 | Variable als Operand 1 gelesen und geschrieben |
| 6 | Variable als Operand 2 gelesen und geschrieben |
| 7 | Variable als Operand 1 und 2 gelesen und geschrieben |

Tabelle 4.2: Matrixbelegung der Variablen

Die in Tabelle 4.2 gezeigte Zuordnung ist exemplarisch gewählt und kann in Implementierungen abweichen. Diese Darstellung ist, sofern notwendig, erweiterbar auf beispielsweise drei Eingangsoperanden. Eine resultierende Matrix, in der die meisten Einträge den Wert Null haben, wird als dünnbesetzte Matrix [Her12] bezeichnet.
Die Besetzungsstruktur, also die von Null verschiedenen Einträge der Matrix geben dabei Aufschluss über die Variablenverteilung im Assemblercode. Bei der Variablenreduktion wird über eine Neuordnung von Zeilen- und Spaltennummern ermittelt, welche Variablen sich in ihren Lebenszeiten überlappen und welche Variablen für eine Reduktion geeignet sind.
Die Lebenszeit einer Variable ist dabei definiert als der Zeitraum zwischen dem ersten Schreiben der Variable und dem letzten Lesen dieser. Bei diesem Verfahren werden Konstanten ebenfalls als Variablen betrachtet, hier zählt die Deklaration als erstes Schreiben.
Sonderfälle, wie beispielsweise Variablen, die nie gelesen werden, werden entsprechend der Definition als Variable ohne Lebenszeit betrachtet. Im Folgenden wird das Beispiel aus Abbildung 4.16 in der dort gezeigten Form für Abbildung 4.21 verwendet, um die Funktionsweise zu demonstrieren.

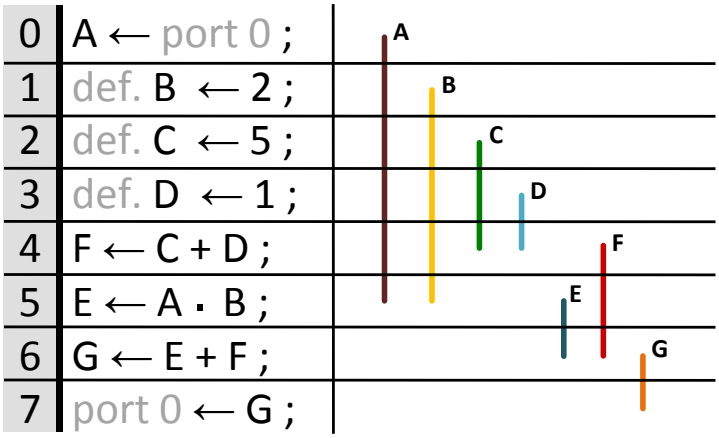



Abbildung 4.21: Variablenlebenszeiten aus Beispiel 4.16

Dieses Beispiel ist exemplarisch gewählt. Normalerweise wird davon ausgegangen, dass die bisher vorgestellten Optimierungsverfahren den Graphen entsprechend vereinfachen, bevor die Variablenreduktion zum Einsatz kommt. Nachdem die Lebenszeiten, dargestellt als farbige Linien in Abbildung 4.21, ermittelt wurden, werden diese nach dem vorgestellten Schema in der Matrix gespeichert. Die Matrix wird nach folgendem Verfahren durchlaufen, dass in Abbildung 4.22 als Pseudocode dargestellt ist.

Input: $[T_0,T_1)$ as period of time, $[s_1,f_1),..,[s_n,f_n)$ as set of lifetimes and $Var_1,..,Var_n$ as variables
Output: $A_1,A_2,..,A_k$ as set of variables with every given lifetime and with $k \leftarrow min$

```
0   sort intervals with f_1,..,f_n ascending;
1   k = 1;
2   Do
3     f_last ← f_first unmarked Var;
4     A_k ← {first unmarked Var} and mark it;
5     for Var_i from Var_1 to Var_n
6       if s_i ≥ f_last and Var_i is unmarked then
7         A_k ← A_k ∪ {Var_i};
8         replace (and mark) Var_i;
9         f_last ← f_i;
10    k = k + 1;
11  till every Var is marked;
12  return A_1, ..., A_k;
```

Abbildung 4.22: Pseudocode zum Durchlaufverfahren

Es ist anzumerken, dass aufgrund des Abstraktionsgrads und dem modularen Gedanken an dieser Stelle keine Informationen über eine spezielle Realisierung aller nachfolgenden Schritte der Toolchain eingehen. Das bedeutet, dass weder bekannt ist, wie groß der verwendete

Variablenspeicher (die Anzahl an verfügbaren Speicherzellen) ist, die der resultierende Prozessor verwendet, wie dieser physisch Adressiert wird, noch ob der Assembler Mechanismen verwendet, die selbst eine Speicherreduktion realisieren.
Im Anschluss an das Generieren der Lebenszeiten (*Step 0*, Abbildung 4.23) werden alle Variablen sortiert nach dem Zeitpunkt, zu dem diese zuletzt gelesen wurden. Das ist in Abbildung 4.23 unter Schritt 1 (*Step 1*) dargestellt.
Die sortierte Liste wird nun iteriert durchlaufen und es wird immer die nächste wählbare Variable gesucht und ersetzt. Variablen werden dabei als ersetzbar angesehen, wenn ihr Lebensstart später oder gleich dem Zeitpunkt des Todes einer bereits markierten Variable liegt.

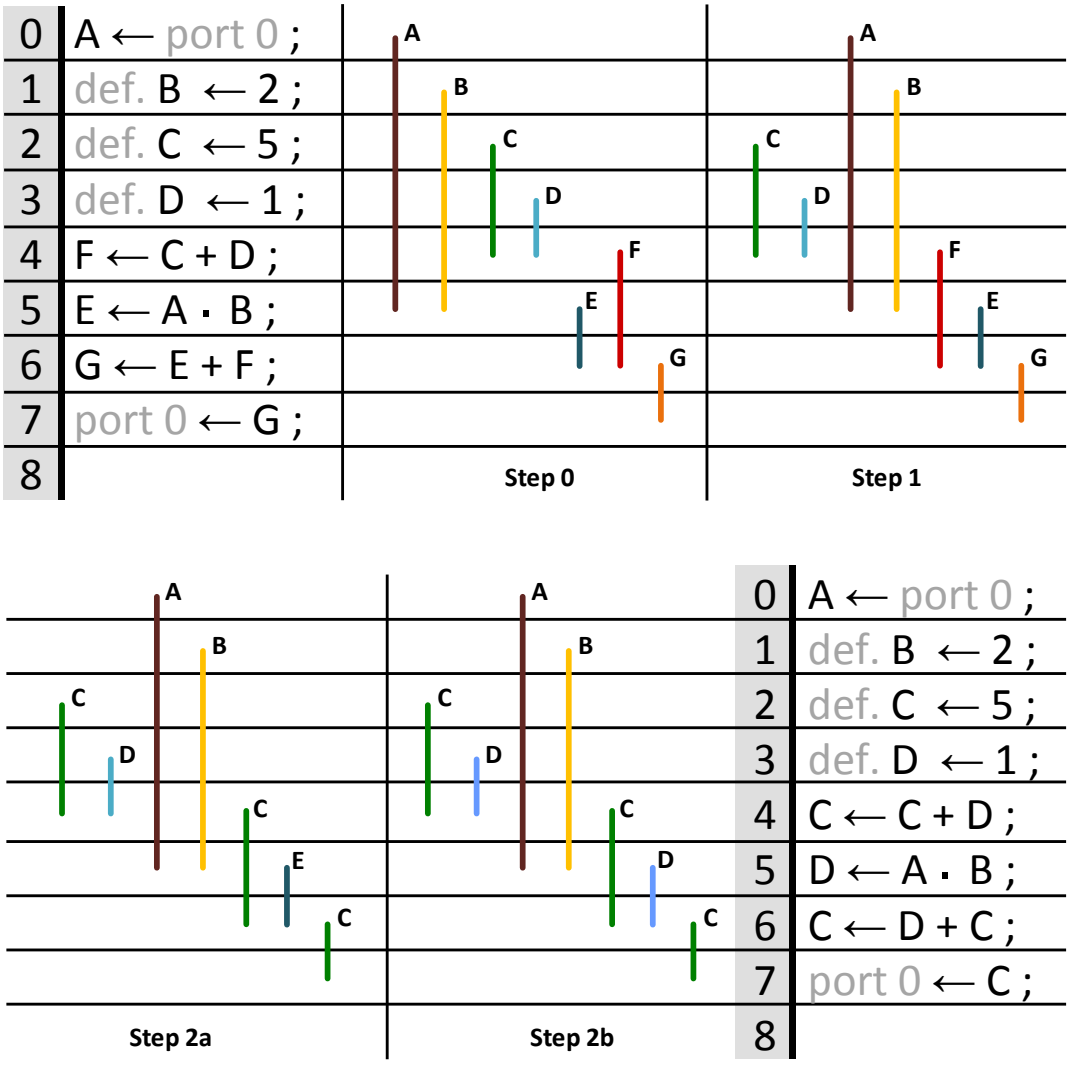



Abbildung 4.23: Reduktion der verwendeten Variablen

Dem Beispiel aus Abbildung 4.16 folgend, kann in Schritt 2a (*Step 2a*) aus Abbildung 4.23 Variable $C$ markiert werden und die nächste ersetzbare Variable wird gesucht. $A$, $B$ und $D$ können nicht ersetzt werden, da sie überschneidende Lebenszeiten besitzen. Die nächste

ersetzbare Variable ist $F$, also wird diese durch $C$ ersetzt. Im Anschluss wird das Verfahren Variable $G$ finden und ebenfalls durch $C$ ersetzen.
In der nächsten Interation wird $D$ markiert und die Matrix wird erneut durchlaufen. In diesem Durchlauf wird $E$ durch $D$ ersetzt. Im weiteren Verlauf gibt es keine weiter ersetzbaren Variablen.
Da sowohl jede Variable, als auch jede Konstante auf dieser Abstraktionsebene einer Speicherzelle entsprechen, kann mit diesem Verfahren die benötigte Speicherauslastung und damit der benötigte Speicher im FPGA minimiert werden.

## 4.6 Entwurf eines optimierenden Assemblers für spezialisierte Softcore Prozessoren

Nachdem ein Assemblercode mit Hilfe der in Abschnitt 4.5 vorgestellten modellbasierten Methodik generiert wurde, muss dieses in für den Softcore lesbaren Maschinencode übersetzt werden. Diese Aufgabe wird von einem optimierenden Assembler übernommen, dessen Methodik in diesem Abschnitt im Detail vorgestellt wird.
Im Wesentlichen ist der Assembler die Schnittstelle zwischen dem Ergebnis der hohen Abstraktionsebene, also dem Assemblercode und dem niedrigen Abstraktionslevel des binären Maschinencodes. Aus diesem Grund ist es wichtig, dass der Assembler eingangsseitig mit der Syntax des Assemblercodes und ausgangsseitig mit der geforderten Syntax des Softcore Prozessors kompatibel ist. Zusätzlich ist es die Aufgabe des Assemblers, wie in Abbildung 4.3 vorgestellt, semantische Metainformationen aus einem gegebenen Assemblercode zu extrahieren, um mit Hilfe dieser Informationen aus einer Softcore-Bibliothek einen spezialisierten Softcore Prozessor möglichst automatisiert generieren zu können.
Der wesentliche Unterschied zwischen dem Assemblercode und dem Maschinencode besteht darin, dass der Assemblercode in dieser Arbeit als eine sequenzielle Abfolge von Befehlen verstanden wird, während ein funktionsgleicher Maschinencode eine potenziell abgeänderte out-of-order Verarbeitung verwendet. Weiterhin sind innerhalb des Maschinencodes, im Gegensatz zum Assemblercode, Informationen über die jeweiligen Operationslaufzeiten enthalten, das bedeutet die Zeitpunkte des Register-lesens und -schreibens sind an unterschiedlichen Stellen im Maschinencode codiert.
Die Einschränkungen, die getroffen werden müssen, um den Kriterien der hier spezifizierten Aufgabendomäne zu entsprechen, vor allem die Einhaltung harter Echtzeitschranken, ermöglichen innerhalb des Assemblers spezielle Lösungen, welche die Effizienz verbessern. Diese Lösungen können bei Assemblern für allgemeine Softcores oder generell Prozessoren nicht verwendet werden, da hier die folgenden Voraussetzungen nicht gelten.
Sowohl der erzeugte Assemblercode, als auch der daraus resultierende Maschinencode müssen zur Designzeit zeitlich voll analysierbar sein. Das bedeutet es kann bereits zur Designzeit ermittelt werden, wie viele Taktzyklen ein Softcore zur Abarbeitung des im Assemblercodes spezifizierten Algorithmus benötigen wird.
Die Verwendung des Assemblers ausschließlich zur Übersetzung von Assemblercodes für FPGA-basierte Softcore Prozessoren schließt spezialisierte Informationen, wie beispielsweise die Verfügbarkeit von exakt definierten arithmetischen Operationen, genauso wie deren Verarbeitungsgeschwindigkeiten und die vorhandene Anzahl an Kernen bei einer Multi-Core-Architektur ein. Daraus folgt die Notwendigkeit, dass der Assembler auf die frequentierten Änderungen dieser Parameter reagieren können muss, da die Spezialisierung und Konfigurierung bei Softcore Prozessoren ein Standard-Use-Case ist. Als Beispiel ist die Änderung der verfügbaren Kerne und

der darauf befindlichen Operatoren zu nennen. Zusätzlich muss der Assembler auf den Einsatz von partiell rekonfigurierbare FPGAs reagieren können.
Das bedeutet, je detaillierter nicht nur die verwendete Softcorearchitektur, sondern auch die spezielle Konfiguration des jeweils verwendeten Softcore als Eingage vom Assembler akzeptiert und berücksichtigt wird, desto besser kann der resultierende Maschinencode optimiert werden, ohne die nötige Flexibilität zu verlieren.
Die im Folgenden vorgestellten matrixbasierten Schedulingalgorithmen sowie die dafür notwendigen Grundlagen wurden bereits in [KKSF17] veröffentlicht.

### 4.6.1 Voraussetzungen der Schedulingalgorithmen

Es wird davon ausgegangen, das der verwendete Softcore Prozessor einen Parallelisierungsmechanismus wie beispielsweise das Prinzip des Aufgabenpipelinings verwendet. Dabei ist es möglich, in jedem Takt neue Operationen zu starten. Die Anzahl an zu startenden Operationen ($OP_{\text{qty}} \in \mathbb{N}$) hängt dabei von der verwendeten Speicherbandbreite, also der Lese- und Schreibgeschwindigkeit, ab und folgt der Gleichung:

$$OP_{\text{qty}} = min\{2 \cdot \frac{read\,operations}{clock\,cycle}, \frac{write\,operations}{clock\,cycle}\} \tag{4.42}$$

Das bedeutet, dass die Anzahl an Operationen, die in jedem Takt in einem voll gepipelinedtem System gestartet werden können, von der doppelten Anzahl an gleichzeitig lesbaren Operanden und der Anzahl an gleichzeitig speicherbaren Ergebnissen abhängt. Wird also davon ausgegangen, dass in jedem Takt ein Ergebnis gespeichert und zwei Operanden gelesen werden können, so ist es möglich in jedem Takt eine neue unabhängige Operation zu starten.
Die Menge der verwendbaren Operationen (*cmd*) ist prozessorabhängig, beinhaltet aber in den meisten Fällen Operationen mit null bis zwei zu lesenden Operanden und einem oder keinem abzuspeichernden Ergebnis. Im Folgenden wird exemplarisch für jeden dieser Fälle ein Beispiel genannt:

| Anzahl Operanden | Anzahl Ergebnisse | Beispiel |
|---|---|---|
| 0 | 1 | Externe Werteingabe |
| 1 | 1 | Wurzel-Operation |
| 2 | 1 | Addition |
| 1 | 0 | Externe Ausgabe eines Wertes |

Tabelle 4.3: Beispiele für mögliche Operationen

Eine Operation $OP_B$ ist abhängig von einer vorhergehenden Operation $OP_A$, wenn $OP_B$ eine lesende oder schreibende Operandenabhängigkeit besitzt. Es können also vier Abhängigkeitsvarianten auftreten, die in Tabelle 4.4 dargestellt sind.

Mit Hilfe dieser vier Abhängigkeitstypen und der Tatsache, dass jeder Operand bis zu zwei lesende und eine schreibende Abhängigkeit besitzen kann, kann ein Abhängigkeitsbaum aufgestellt werden. Dieser wird im Weiteren als Levelbaum (*Level Tree*) bezeichnet.
Es gilt die Annahme, dass es einen Quellknoten $S$ auf Level 0 gibt, der den Beginn des Programms darstellt. Weiterhin gilt die Annahme, dass jede Operation $M$ von eben diesem Quellknoten

| | |
|---|---|
| read-after-read (RaR) | read-after-write (RaW) |
| write-after-read (WaR) | write-after-write (WaW) |

Tabelle 4.4: Abhängigkeitsvarianten der Operationen

abhängig ist ($S \rightarrow M$). Jeder Operation kann mit diesen Annahmen und der folgenden Vorschrift ein Level zugeordnet werden:

$$level_M = \begin{cases} level_K + 1 & : \ K \rightarrow M, \\ level_L + 1 & : \ K, L \rightarrow M \wedge \ level_L > level_K. \end{cases} \tag{4.43}$$

Das bedeutet, dass $level_M$ gleich dem Level von Knoten $K$ plus eins ist, wenn $M$ von $K$ abhängig ist (oder: aus $K$ folgt $M$), oder dem Level von $L$ plus eins, wenn $M$ sowohl von $K$, als auch von $L$ abhängig ist und das Level von $L$ größer dem Level von $K$ ist ($level_L > level_K$).
Mit Hilfe dieses Graphen ist es möglich, eine an die Aufgabenklasse modifizierte Adjazenzmatrix zu erstellen:
Im Allgemeinem symbolisiert eine Adjazenzmatrix einen direktionalen Kontrollflussgraphen. Das bedeutet, eine Adjazenzmatrix ist eine $n \times n$ Matrix $A$, in der $A[i, j]$ die Kante zwischen den Knoten $i$ und $j$ definiert. Dabei ist der Wert von $A[i, j]$ gleich 1, wenn es eine verbindende Kante zwischen $i$ und $j$ gibt, und 0 andernfalls. [Bap14]
Für die methodische Definition des hier erläuterten Assemblers steht $n$ für die Nummer an Assemblerbefehlen in dem spezifizierten Programm. Weiterhin repräsentiert diese Matrix die Abhängigkeiten zwischen den verschiedenen Assemblerbefehlen. Entsprechend muss die Adjazenzmatrix wie folgt modifiziert werden: Jedes Element $A[i, j]$ repräsentiert die Level-Differenz zwischen zwei Knoten $i$ und $j$:

$$A[i, j] = level_j - level_i \tag{4.44}$$

Mit Hilfe dieser Anpassung kann eine Aussage getroffen werden, ob ein Befehl $j$ von einem anderen Befehl $i$ abhängig ist. Dieser Fall kann nur auftreten, wenn gilt $j > i$ und ist gleichbedeutend mit der Aussage, dass der Befehl $j$ nach dem Befehl $i$ ausgeführt werden muss und beide Befehle teilen sich mindestens eine gemeinsame Speicherzelle.
Die Informationen aller Knoten, die von $j$ abhängen, befindet sich also in Spalte $j$ und alle Abhängigkeiten von $i$ werden in Reihe $i$ dargestellt. Dank dieser Anpassung ist es nur notwendig, die gerichtete Abhängigkeit von Knoten $i$ zum Knoten $j$ (oder $i \rightarrow j$) des Levelbaums in der Adjazenzmatrix zu betrachten.

Eine weitere notwendige Matrix stellt die Inzidenzmatrix dar. Im Allgemeinen ist eine Inzidenzmatrix eine $n \times m$ Matrix $B$, in der $B[i, j]$ den Wert 0 annimmt, wenn der Knoten $i$ und die Kante $e_j$ nicht inzident sind. Der Wert wird 1 bzw. -1, wenn der Knoten und die

Kante inzident sind. Dabei ist das Vorzeichen abhängig von der Richtung von $e_j$ in Relation zu $i$. [Bap14]
In der hier vorgestellten Methodik wird die Inzidenzmatrix des Assemblers verwendet, um die Relation zwischen den Assemblerbefehlen und den darin verwendeten Speicherzellen darzustellen. Ein Befehl kann von bis zu drei Speicherzellen, zwei zur Eingabe und eine zur Ausgabe verwendete Speicherzelle, abhängig sein. Das bedeutet, in diesem Fall stellt bei der $n \times m$ Matrix $B$ $n$ die Anzahl der Assemblerbefehle und $m$ die Anzahl der benötigten Speicherzellen dar.
Aufgrund der möglichen verschiedenen Abhängigkeiten muss die Matrix allerdings angepasst werden:
Bei jeder neuen Operation, also jedem neuen Assemblerbefehl, benötigen (bis zu) zwei Speicherzellen einen lesenden Zugriff und (bis zu) eine Speicherzelle einen schreibenden Zugriff. Um den schreibenden vom lesenden Zugriff differenzieren zu können wird festgelegt, dass jedes Element $B[i, j]$ einen positiven Wert für den lesenden und einen negativen Wert für den schreibenden Zugriff erhält. Weiterhin wird der (vorzeichenunabhängige) Wert aus dem Parameter *destination met* gebildet. Dieser Parameter repräsentiert die aktuelle Anzahl an Änderungen eines Wertes einer speziellen Speicherzelle und wird als negative Zahl ausgedrückt. Wird eine Speicherzelle $j$ beispielsweise zweimal gelesen und weitere zweimal geschrieben, so ist $destMet[j] = -4$. Der Parameter wird sich über die Berechnungszeit ändern. Das bedeutet, zu jedem Zeitpunkt wird diese spezielle Speicherzelle entsprechend oft durch ihre Vorgängerknoten in ihrem Wert verändert und der Parameter wird inkrementiert, sobald ein Vorgängerknoten abgearbeitet ist.
Das heißt, wenn eine Speicherzelle $j$ ihren Wert nach der Ausführung von dem Befehl $i$ ändert, wird $destMet[j]$ inkrementiert. Also ist der Wert von $|destMet[j]|$ abhängig von der Anzahl an Lese- und Schreibzugriffen auf die Speicherzelle $j$ und ändert sich mit der Ausführung des Algorithmus. Diese Regeln ergeben die geltende Gleichung:

$$B[i,j] = \begin{cases} |destMet[j]| & \text{,wenn } i \text{ } j \text{ liest,} \\ destMet[j] & \text{,wenn } i \text{ } j \text{ schreibt.} \end{cases} \tag{4.45}$$

Sobald eine Operation $i$ beendet ist und das Ergebnis in der dafür vorgesehenen Speicherzelle gespeichert wurde, wird jede weitere Operation $i+1$, die lediglich von der Operation $i$ abhängig gewesen ist, als unabhängig betrachtet und kann gestartet werden, was eine Adaption der Matrix bedeutet. In diesem Fall hat der Parameter $destMet[j+1]$ von der Operation $i+1$ den Wert 0.

### 4.6.2 Speicherverwaltung

Der einfachste Ansatz im Umgang mit Variablen sieht vor, dass jede Variable während der gesamten Abarbeitungszeit des Algorithmus an eine spezifische Speicherzelle, Speicheradresse oder Register gebunden ist. Dieser Ansatz führt zu verschiedenen Problemen in Punkten wie Effizienz und Skalierbarkeit. Eine Variable könnte beispielsweise nur ein oder zwei Mal innerhalb eines Programms verwendet werden. Das führt zu sehr schlecht ausgelasteten Speicherzellen und resultiert in einem ineffizienten Speichermanagement. In einem weiteren Szenario könnten mehrere Operationen die gleiche Variable verwenden. Die daraus entstehenden Abhängigkeiten verhindern, dass der Assembler die Verarbeitungsgeschwindigkeit erhöhen kann, indem dieser

die Pipelineauslastung maximiert.

Diese Probleme treten normalerweise bei Superskalaren Out-of-Order Prozessoren auf. Dabei handelt es sich um Prozessoren, die mehrere Befehle aus einem Befehlsstrom gleichzeitig mit parallel arbeitenden Funktionseinheiten verarbeiten können. Out-of-Order bedeutet in diesem Fall, dass die Befehle vom Backend potenziell in einer anderen Reihenfolge ausgeführt werden, als sie vom Frontend durch Instruction-Fetch, Branch Prediction und Dekodierung vorgegeben werden.
Hierbei kommt ein Verfahren zum Einsatz, dass WaR- und WaW-Abhängigkeiten im Prozessor auflösen kann. Dieses Verfahren wird als Register Renaming [Ung01] bezeichnet.
Bei diesem Verfahren bleiben alle Register virtuelle Register bis zum Zeitpunkt der Fertigstellung der jeweiligen Befehle. Zu diesem Zeitpunkt wird jedes virtuelle Register auf ein physisches Register gemappt. Das bedeutet, dass das Ergebnis der Operation auf einem zu diesem Zeitpunkt freien physischen Register gespeichert wird. Freie Register können dabei beispielsweise einfach als gespeicherte Liste vorliegen. [Bae10]

Aufgrund der in der vorliegenden Arbeit adressierten Problemklassen ist es nicht sinnvoll, den Prozessor eine Auflösung entsprechender Abhängigkeiten zur Laufzeit vornehmen zu lassen. Das würde dafür sorgen, dass eine Laufzeitanalyse zur Designzeit stark erschwert würde. Meist wird die Lösung dieses Problems von einem Compiler übernommen. In dieser Arbeit wird vorgeschlagen, dieses Verfahren anzupassen und in den Assembler (anstatt den Compiler) zu integrieren. Dadurch erfolgt das Mapping aller physisch vorhandenen Speicherzellen mit im Assemblercode verwendeten Variablen zur Designzeit und ist damit auch voll analysierbar.
Ein weiterer Vorteil, im Vergleich zum eigentlichen Register Renaming, liegt darin, dass dieser Vorgang zur Designzeit und nicht dynamisch zur Laufzeit ausgeführt wird. Dadurch ist das Suchen und Finden am besten geeigneter Speicherslots kein zeitkritisches Problem. Der Assembler hat daher theoretisch sogar die Möglichkeit, den gesamten Speicher für jede Variable auf der Suche nach dem optimalen Speicherslot erneut zu durchlaufen, ohne das es zu einer Verletzung einer Laufzeit kommt.
Es soll innerhalb des Assemblers für Variablenwerte das gleiche Konzept verwendet werden, wie für Befehle: Eine Variable wird als eine Klasse angenommen und die eigentlichen Variablenwerte (die Werte, die eine semantische Variable annehmen kann) werden als Instanzen dieser Klasse (*Variable-Object*) angesehen. Diese Idee wurde bereits durch die in dieser Arbeit verwendeten Multi-Graphen aufgegriffen und wird im Assembler weitergeführt.
So sind beispielsweise zwei gleiche Operationen nicht voneinander abhängig, nur auf Grund der Tatsache das beide die gleiche mathematische Berechnung ausführen, da jeder Operator eine interne Pipeline verwendet. Wird dieser Ansatz auch auf Variablen und deren mögliche Variablenwerte angewendet, kann die Verarbeitungsgeschwindigkeit von Programmen stark erhöht werden. Dieser Ansatz kann noch erweitert werden, indem jede individuelle Belegung jeder Variable, also jede im Softcore für eine semantische Variable verwendete Speicherzelle als individuelle Instanz durch ein Variable-Wert-Objekt (*Variable-Value-Object*) dargestellt werden kann. Mit diesem Ansatz werden aus Variablen sogenannte „Factory-Objects“ deren Aufgabe es ist für jede individuell benötigte Instanz die entsprechenden Werte bereitzustellen.
Der Ansatz basiert auf der Annahme, dass Assemblercode durch ein Modellierungstool auf einer höheren Abstraktionsebene erstellt und im Anschluss automatisch generiert wurde und die Variablen deswegen ineffizient, im Sinne der maximalen Pipelineauslastung der Operatoren, zugeordnet worden. Alternativ wird ebenfalls angenommen, dass die Variablen bei händisch geschriebenen Assemblercode nach logisch zusammenhängenden Blöcken zugeordnet wurden,

was ebenfalls einer für eine maximale Pipelineauslastung ineffizienten Zuordnung entspricht. Ein einfaches Beispiel, dass das Prinzip verdeutlicht, wird in Abbildung 4.24 vorgestellt:

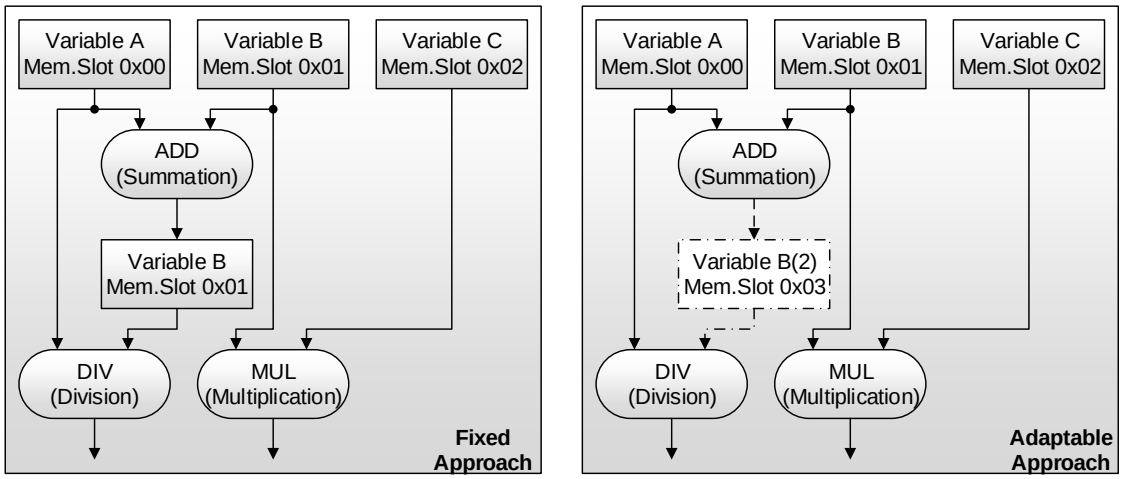



Abbildung 4.24: Beispiel für ein Variable-Wert-Objekt

Zu sehen sind drei Variablen $A$, $B$ und $C$, die in verschiedenen arithmetischen Operationen verwendet werden. Die Addition (*Summation*) verwendet $A$ und $B$, die Division (*Division*) verwendet das Ergebnis der Addition, gespeichert in $B$ und den originalen Wert von $A$, und die Multiplikation (*Multiplication*) verwendet den originalen Wert von $B$ und $C$. Wie deutlich zu sehen ist, gibt es eine problematische Abhängigkeitskette zwischen den drei Operationen, wenn der traditionelle Ansatz (*Fixed Approach*) verwendet wird: Die Multiplikation muss gestartet werden, bevor die Addition den originalen Wert der Variable $B$ überschreibt, aber die Division benötigt das Ergebnis der Addition bevor diese gestartet werden kann. Es existiert also eine RaW-Abhängigkeit zwischen der Division und der Addition sowie eine WaR-Abhängigkeit zwischen der Addition und der Multiplikation.
Das führt zu einer ineffizienten Verwendung der Pipeline, da der Assembler gezwungen ist direkt hintereinander liegende Zeitslots zu finden, in denen beide Operationen, also Addition und Multiplikation, direkt nacheinander gestartet werden können, oder der Assembler ist sogar gezwungen die Multiplikation vor der Addition zu starten. Diese Vorgehensweise ist ineffizient, da die Addition später gestartet wird und infolgedessen auch die davon abhängige Division entsprechend verzögert werden würde. Das Resultat ist eine verlängerte Rechenzeit für das gegebene Programm. Diese Probleme werden als Hazards bezeichnet und wurden bereits im Abschnitt 2.5.2 vorgestellt.
Wird der hier vorgestellte Ansatz (*Adaptable Approach*) verwendet, ist es dem Assembler möglich den neuen Wert der Variable $B$ in einer anderen Speicherzelle (in diesem Beispiel Zelle *0x03*) zu speichern und damit ein weiteres Variable-Wert-Objekt der Variable $B$ zu erzeugen. Mit Hilfe dieses neuen Objekts können nun die beiden ersten Operationen, also Addition und Multiplikation, komplett unabhängig voneinander gescheduled werden, was die Möglichkeit für den Assembler eröffnet, den jeweils bestmöglich geeigneten Zeitslot für jede Operation zu wählen und damit die gesamte Rechenzeit des gegebenen Programms zu minimieren.
Um also eine optimale Nutzung der zur Verfügung stehenden Speicherslots zu gewährleisten, wird jeder lesende und schreibende Zugriff auf eine Variable in dem zugehörigen Wert-Objekt (*Value-Object*) registriert. So ist es möglich zu bestimmen, wann ein bestimmter Wert einer Variable produziert (geschrieben) oder konsumiert (gelesen) wird. Wird eine Multi-Core-Architektur

verwendet, werden zusätzlich Informationen gespeichert, welcher Kern den jeweiligen Wert geschrieben hat und ob dieser Wert zusätzlich in einem gemeinsamen Speicher, oder dem lokalen Speicher eines anderen Kerns abgelegt wurde. Das bedeutet, sowohl die WaR-Hazards, als auch die WaW-Hazards werden mit diesem Verfahren adressiert.
Der wesentliche Unterschied zu anderen bereits existierenden Verfahren der Hazarddetektion und Behebung, wie beispielsweise in [CM03] vorgestellt, liegt darin Variablen als instantiierbare Objekte mit zugehörigen Wert-Objekten zu betrachten, wie bereits erläutert, und diese in den Assembler anstatt einen übergeordneten Compiler zu integrieren.
Diese zusätzliche Komplexität innerhalb des Assemblers eröffnet die Möglichkeit, die jeweiligen Variablen deutlich genauer verarbeiten zu können, ergibt eine bessere Handhabung von Abhängigkeiten zwischen verschiedenen Operationen und resultiert in einer Verbesserung der Parallelität der Instruktionsabarbeitung innerhalb des Softcore Prozessors. Wenn jeder mögliche Zustand jeder Variable zu jedem Zeitpunkt individuell verfolgt wird und mit den Informationen wann ein Wert produziert und wann das letzte Mal konsumiert wird, ist es möglich eine individuelle Speicherzelle für jede dieser Variablenwerte zu verwenden. Das Ergebnis ist eine Speicherstruktur, in der keine Variable an eine feste Speicherzelle gebunden ist.
Es ist dadurch weiterhin möglich, alte nicht mehr benötigte Variablen, bzw. dessen Speicherzellen, mit neuen Variable-Wert-Objekten anderer Variablen zu überschreiben. Diese Mechanismen maximieren die effektive Speicherauslastung.
Das führt zu zwei möglichen Vorteilen, je nach der Art des zu übersetzenden Programms durch den Assembler:

1. Werden innerhalb eines Programms eine Vielzahl von Variablen deklariert und diese nur wenig durch Operationen konsumiert, werden verschiedene Variablen durch den Assembler auf dieselbe Speicherzelle gemappt, was eine deutlich reduzierte Anzahl an verwendeten Speicherzellen entspricht. Der Assembler wird dabei über die Lebenszeiten der Variable-Wert-Objekte sicherstellen, dass die Integrität jeder Variable sichergestellt ist.

2. Wenn innerhalb eines Programms eine Vielzahl von voneinander abhängigen Operationen verwendet werden, ist es möglich mit diesem Ansatz die Parallelität des resultierenden Maschinencodes deutlich zu erhöhen, indem eine Variable mit unterschiedlichen Wert-Belegungen in verschiedenen Speicherzellen abgespeichert wird. Das ist vorteilhaft für eine Ein-Kern-Architektur, aber auch für Mehr-Kern-Architekturen, da hier die einzelnen Werte entsprechend ihrer Verwendung entweder lokal auf verschiedenen Kernen gespeichert werden können (wo sie benötigt werden), aber mit anderen Werten dennoch auf anderen Kernen verwendet, als auch innerhalb eines gemeinsamen Speichers zur gemeinsamen Verwendung abgelegt werden können. Das resultiert in einem verringerten Kommunikationsoverhead der einzelnen Kerne. Dabei ist das Speichern einer Variable in mehreren Speicherzellen kein Nachteil, da es aufgrund der Verwendung von Softcore Prozessoren innerhalb von FPGAs eine zur Designzeit definierte Anzahl an verfügbaren Speicherzellen geben muss und diese dadurch lediglich optimaler ausgenutzt werden.

### 4.6.3 Matrizenbasierte Schedulingalgorithmen

Auf Basis der vorgestellten modifizierten Matrizen und des Levelbaums können Verfahren entwickelt werden, die ein effizientes Scheduling beim Übersetzen von Assemblercode in für den Softcore lesbaren Maschinencode garantieren. Effizient heißt in diesem Zusammenhang, dass sich das Ergebnis des Schedulings, in Bezug auf gestartete Operationen je Takt, dem Wert

aus Gleichung 4.42 berechneten theoretischen Maximum annähert, ohne dabei die Funktion des zugrunde liegenden Assemblercodes zu verändern. Das bedeutet, je näher der erzielte Wert an dem theoretischen Maximum liegt, desto effizienter sind die jeweiligen Schedulingverfahren. Damit ist die Effizienz der durchschnittlichen Prozessorauslastung gemeint. Eine Steigerung der Effizienz resultiert in einer Minimierung des Verarbeitungszeit des zugrunde liegenden Assemblercodes.
Da jedes zu übersetzende Programm spezielle Eigenschaften, zusätzlich zu den durch die Aufgabenklasse spezifizierte Eigenschaften, besitzt, ist es sinnvoll verschiedene Schedulingalgorithmen zu verwenden. Es wird dabei von der Annahme ausgegangen, dass **P≠NP** gilt und es deswegen nicht möglich ist mit polynomieller Rechenzeit in sehr großen Graphen die gesamte Anzahl an möglichen optimalen Wegen zu finden, und damit alle optimalen Schedulings zu finden. [PCCC05]
Aus diesem Grund müssen bei jedem Schedulingalgorithmus vereinfachende Annahmen über die Eigenschaften des vorliegenden Problems getroffen werden. Diese einzelnen Algorithmen zielen damit auf jeweils unterschiedliche Charakteristika des Programmcodes ab und sind von Fall zu Fall unterschiedlich effektiv. Deshalb ist es sinnvoll, eine Variation an Schedulingalgorithmen zur Verfügung zu haben um jeweils ein sehr gutes Ergebnis erzielen zu können.
Nachfolgend werden deshalb insgesamt vier matrixbasierte Optimierungsalgorithmen vorgestellt und diskutiert. Für alle Algorithmen gilt, dass diese unabhängige Programmstücke finden müssen. Ein Programmstück ist dabei linear unabhängig, wenn die folgenden Bedingungen erfüllt sind:

- RaW-Unabhängigkeit: Programmstück $B$ benötigt keinen lesenden Zugriff auf Speicherzellen, die ihre Werte im Verlauf der Ausführung von Programmstück $A$ verändern.
- WaR-Unabhängigkeit: Programmstück $A$ benötigt keinen schreibenden Zugriff auf Speicherzellen, die in Programmstück $B$ gelesen werden.
- WaW-Unabhängigkeit: Programmstück $B$ benötigt keinen schreibenden Zugriff auf Speicherzellen, die in Programmstück $A$ verwendet werden.

Lediglich die RaR-Abhängigkeit zweier Programmstücke $A$ und $B$ ist erlaubt, damit diese als linear unabhängig gelten.
Es gibt verschiedene berechenbare Größen, mit denen die Effizienz eines Schedulings dargestellt werden kann. Mit der Prozessorauslastung wurde eine Größe bereits vorgestellt und wird im Folgenden genau definiert. Eine weitere messbare Größe ist die resultierende Rechenzeit für ein gegebenes Problem. Um die Effizienz in Bezug auf der resultierenden Rechenzeit der Schedulingalgorithmen messen bzw. berechnen zu können, muss zuerst die nicht optimierte (und damit maximale) Rechenzeit des Programms ermittelt werden:

$$maxTime = \sum_{i=0}^{maxCmd-1} (aluDelay_i + pipelineDelay_i) \tag{4.46}$$

Wobei $aluDelay_i$ die Rechenzeit jeder Operation und $pipelineDelay_i$ die Zeit von der Verfügbarkeit des Ergebnisses bis zum frühesten Zeitpunkt, an dem dieses Ergebnis als Operand verwendet werden kann, ist. $maxCmd$ stellt die gesamte Anzahl an Operationen dar, die das spezielle Programm verwendet.
Mit diesem Wert kann die zeitliche Effizienz ($E_t$), also die prozentuale Rechenzeit, des jeweiligen Schedulings ausgerechnet werden:

$$E_t = \frac{time \cdot 100}{maxTime} \tag{4.47}$$

Wobei *time* die benötigte Rechenzeit eines gegebenen Schedulings ist. Sowohl *time* als auch $maxTime$ werden dabei in Takten angegeben.
Die Prozessorauslastung je Takt (*workload*) ist das prozentuale Verhältnis der Anzahl von Operationen ($OP_\mathrm{i}$), die in einem bestimmten Takt gestartet werden zum theoretischen Maximum ($OP_\mathrm{qty}$) der Anzahl an Operationen, die in jedem Takt gestartet werden könnten:

$$workload = \frac{OP_i \cdot 100}{OP_\mathrm{qty}} \tag{4.48}$$

Daraus ergibt sich die durchschnittliche Prozessorauslastung $E_p$ wie folgt:

$$E_\mathrm{p} = \frac{\sum_{i=0}^{time-1} (workload_i)}{time} \tag{4.49}$$

Mit Hilfe der Rechenzeit ($E_t$) und der durchschnittlichen Prozessorauslastung ($E_p$) lassen sich alle Schedulings miteinander vergleichen. Die durchschnittlichen Prozessorauslastung wird dabei auch als Pipelineauslastung bezeichnet.

#### Algorithmus des maximalen Delays

Der erste präsentierte Algorithmus arbeitet nach dem Prinzip des maximalen Delays. Das bedeutet, dass zu jedem Takt aus dem Pool an wählbaren Operationen immer die Operation gewählt wird, welche die längste Rechenzeit hat.
Die Idee hinter diesem Algorithmus ist, dass Operationen die eine lange Rechenzeit benötigen zuerst gestartet werden sollten, damit die Ergebnisse dieser Berechnungen zum frühest möglichen Zeitpunkt vorliegen. Dabei wird die zu einem Zeitpunkt längste startbare Operation als $OP_\mathrm{long}$ bezeichnet. Dadurch liegen zum Zeitpunkt der Fertigstellung der Operation bereits sowohl das Ergebnis aus eben dieser vor, als auch die Ergebnisse der schneller berechneten Operationen, die nach $OP_\mathrm{long}$ gestartet wurden. Dadurch wird der Pool an Ergebnissen zum Zeitpunkt der Fertigstellung der rechenaufwendigen Operationen maximiert und etwaige von den Ergebnissen aus mehreren Operationen abhängige Folgeoperationen können früher gestartet werden. Auch Folgeoperationen, die lediglich von $OP_\mathrm{long}$ abhängig sind, können so zu einem füheren Zeitpunkt berechnet werden.
Dieser Algorithmus wird also effizientere Schedulings erzeugen, wenn der Großteil der Operationen wiederum von Operationen abhängig ist, die selbst eine sehr lange Rechenzeit benötigen.
Sind allerdings Folgeoperationen lediglich von Operationen mit kurzen Berechnungszeiten abhängig, können diese erst tendenziell später gestartet werden, was zu einer Verlängerung der Rechenzeit des gesamten Programms führen kann.

### Algorithmus des minimalen Delays

Der zweite Algorithmus ist das Gegenstück des ersten Algorithmus. Im Gegensatz zu diesem wird hier immer die Operation ausgewählt, welche die kürzeste Rechenzeit aus dem Pool an startbaren Operationen besitzt. Die Argumentation hinter diesem Schedulingalgorithmus begründet sich aus folgender Annahme: wenn zu jedem Zeitpunkt immer die Operation mit der kürzest möglichen Rechenzeit gewählt wird, liegen nach einem bestimmten Zeitintervall mehr Ergebnisse aufgrund der kürzeren Rechenzeiten vor und dadurch können mehr davon abhängige Operationen in den Pool von startbaren Operationen aufgenommen werden. Wichtig hierbei ist, dass die Abhängigkeiten sich fast ausschließlich auf Operationen mit kurzen Rechenzeiten beschränken, denn jede Abhängigkeit zu einer Operation mit sehr langen Rechenzeiten wird in diesem Algorithmus erst sehr spät aufgelöst werden.
Dieser Algorithmus wird also effizientere Schedulings erzeugen, wenn der Großteil der Operationen wiederum von Operationen abhängig ist, die selbst eine sehr kurze Rechenzeit benötigen.
Der Nachteil wird erkennbar, wenn man sich Assemblerprogramme ansieht, deren Operationen große Differenzen in den Berechnungszeiten aufweisen, da hier lediglich lokale Abhängigkeiten betrachtet werden.

### Algorithmus der minimalen Distanz

Bei dem Scheduling nach der minimalen Distanz wird eine Reihenfolge der Operationen nach der Differenz im Level ($level$), wie durch den Levelbaum und durch Gleichung 4.43 festgelegt. Mit Hilfe der Inzidenzmatrix durchsucht der Algorithmus alle Knoten $i$ auf nachfolgende Knoten $j$, für die gilt:

$$dif_{i,j} = level_j - level_i \rightarrow min \tag{4.50}$$

Mit Hilfe dieser Informationen werden Operationen zu Zeitslots zugeteilt nach dem Verfahren:

$$timeslot_k = \begin{cases} op_i & : \ node_i \text{ unabhängig} \wedge dif_{i,j} = min, \\ \text{slot bleibt frei} & : \ \text{sonst.} \end{cases} \tag{4.51}$$

Der Vorteil dieses Verfahrens liegt darin, dass zu einem möglichst frühen Zeitpunkt die Abhängigkeiten von startbaren Operationen zu davon abhängigen Operationen aufgelöst werden, indem immer die Operationen mit einer hohen Priorität gestartet werden, die entsprechende Abhängigkeiten besitzen. Dabei erfolgt die Auswahl der Operationen unabhängig von der Art und ihren sonstigen Eigenschaften wie beispielsweise der benötigten Rechenzeit. Dadurch wird sichergestellt, dass zu jedem Zeitpunkt die Menge an theoretisch startbaren Operationen maximiert wird.

### Der Time-Distance Ratio Algorithmus

Der Time-Distance Ratio Algorithmus ($TDR$) zielt darauf ab, die Vorteile der bisher vorgestellten Algorithmen zu verbinden und ihre bei gewissen Problemklassen auftretenden Nachteile zu minimieren. Dabei wird die Relation von Zeit und Distanz von einem Knoten, also einer

Operation, zu dessen Nachfolgeknoten verwendet. TDR wird für jeden Knoten $i$ wie folgt berechnet:

$$TDR_i = (\sum_{j=level_i}^{level_k} averageDelay_j) - delay_i \tag{4.52}$$

Wobei $level_k$ das Level des jeweiligen Nachfolgeknotens von $i$ bezeichnet und $averageDelay_i$ entspricht dabei der Gleichung 4.54. $delay_i$ ist die knotenspezifische Rechenzeit wie in Gleichung 4.53 dargestellt, die die Operation benötigt, welche vom Knoten $i$ repräsentiert wird.

$$delay_i = aluDelay_i + pipelineDelay_i \tag{4.53}$$

$$averageDelay_j = \frac{\sum_{m=0}^{cmdNumber} levelDelay_m}{nodesPerLevel_i} \tag{4.54}$$

Dabei gilt:

$$levelDelay_m = \begin{cases} delay_i & : \ m \in level_i, \\ 0 & : \ m \notin level_i. \end{cases} \tag{4.55}$$

$nodesPerLevel_i$ aus Gleichung 4.54 entspricht der Anzahl an Knoten, die $level_i$ haben, und $cmdNumber$ ist die gesamte Anzahl an Knoten im Graphen. Das Ergebnis der Berechnung von $TDR_i$ kann dabei entweder positiv oder negativ sein. Die Schedulingreihenfolge wird durch die Werte von $TDR_i$ bestimmt, dabei werden die Knoten mit dem kleinsten Wert zuerst eingeordnet, um die gesamte Rechenzeit des gegebenen Programms zu minimieren. Hierbei wird sowohl auf die Rechenzeit ($delay_i$) der jeweiligen Operationen Rücksicht genommen, aber auch auf die jeweiligen durch interne Abhängigkeiten entstehende Levelunterschiede innerhalb des Levelbaums. Das resultiert in einer Minimierung der qualitativen Veränderung des Schedulingergebnisses aufgrund von programmspezifischen Charakteristika.

### 4.6.4 Penaltybasierte Schedulingalgorithmen

Die in Abschnitt 4.6.3 vorgestellten Schedulingalgorithmen treffen aufgrund der Problemgröße verschiedene Annahmen über Eigenschaften der Programme, um eine Verbesserung im resultierenden Scheduling zu erzielen. Ein Beispiel ist die Annahme, dass der Großteil der Operationen wiederum von Operationen abhängig ist, die selbst eine sehr lange Rechenzeit benötigen.
Da das Ergebnis aber stark davon abhängt, inwiefern diese Annahmen bei jedem speziellen Programm auch zutreffen, die Schedulingalgorithmen sich allerdings nicht anpassen können, kann die Qualität der Ergebnisse von Programm zu Programm variieren. Im Weiteren wird ein Ansatz vorgestellt, der dynamischer auf die jeweiligen Eigenschaften der Programme eingehen und daher bessere Ergebnisse erzielen soll.
Um einen bestmöglichen Schedulingalgorithmus für jedes Programm in Kombination mit der jeweils verwendeten Hardwarearchitektur verwenden zu können, wird bei den penaltybasierten

Schedulingalgorithmen eine Kombination aus verschiedenen Metriken zur Festlegung der Schedulingreihenfolge verwendet.
Dabei kommt die Idee eines objektorientierten Ansatz zum Einsatz um die notwendige Funktionalität gewährleisten zu können. Dabei ist es dem Nutzer möglich, entsprechende Bestrafungsausdrücke (*Penalties*) zu definieren, die vom Assembler analysiert werden und in einem Penalty-Calculation Objekt gespeichert werden. Das Scheduling wird dann auf Basis dieses Objekts ausgeführt und ist damit nicht mehr an fest vorgegebene Algorithmen gebunden. Der Assembler wird mit Hilfe dieses Ausdrucks für jede Operation zu jedem möglichen Zeitslot einen Bestrafungswert (*Penaltiy Value*) berechnen und die Operationen nach vom geringsten Bestrafungswert aufsteigend sortieren und schedulen.
Diese Metriken können dabei einfache Ausdrücke beinhalten, wie beispielsweise die Rechenzeit der jeweiligen Operation, über die Verfügbarkeit der zu lesenden Speicherzelle auf einem bestimmten Kern bis hin zu komplexen Ausdrücken wie beispielsweise die totale Summe der Anzahl an lesenden Operationen des Ergebnisses unter allen verfügbaren Kernen:

$$Result.ReadOperations.Total = \sum_{i=0}^{CoreNumber-1} Result.ReadOperations.Core.i \tag{4.56}$$

Mit Hilfe dieser Ausdrücke können dann ebenfalls die zuvor vorgestellten matrixbasierten Schedulingalgorithmen nachgebaut werden. Beispielsweise würde die Metrik der benötigten Rechenzeit ausreichen, um die Algorithmen des minimalen und des maximalen Delays nachzubilden.
Weitere als Metriken mögliche zu berücksichtigende Werte:

- Rechenzeit der Operation
- Verfügbarkeit des zu lesenden Inhalts der Speicherzelle in Abhängigkeit von anderen Operationen
- Verfügbarkeit des zu lesenden Inhalts der Speicherzelle auf dem Kern
- Verfügbarkeit des zu lesenden Inhalts der Speicherzelle auf dem gemeinsamen Speicher
- Verfügbarkeit des zu lesenden Inhalts der Speicherzelle auf einem Ergebnisbypass
- Totale Summe der Anzahl an lesenden Operationen vom Ergebnis der Operation
- Verzögerung beim Abspeichern des Ergebnisses
- Anzahl der Operationen, die von der aktuellen Operation abhängig sind

Diese Liste kann, je nach Anwendungsgebiet, Aufgabenklasse und spezieller Eigenschaften des realisierten Softcore Prozessors, erweitert werden. Weiterhin werden mögliche Metriken von den speziellen Eigenschaften der definierten Assemblersprache beeinflusst.
Ein weiterer Vorteil bei der Verwendung von penaltybasierten Schedulingalgorithmen liegt darin, dass diese einfach verständlich sind und nicht so komplex, wie beispielsweise der TDR-Algorithmus. Eine einfache Aneinanderkettung von Metriken ist notwendig, um einen komplett neuen auf die Problemstellung und spezielle Charakteristik angepassten Schedulingalgorithmus zu erstellen.

# 5 Fallstudie zur praktischen Anwendung der modellbasierten Entwurfsverfahren

In diesem Kapitel werden die mit Hilfe der bisher erläuterten Methodik praktische Realisierungen der einzelnen Schritte vorgestellt. In diesem Zusammenhang werden realisierungstechnische Details aufgezeigt und deren Vorteile genannt. Etwaige weitere Tools, die zum Testen oder der Auswertung benötigt werden, werden in den entsprechenden Abschnitten ebenfalls vorgestellt. Die jeweiligen Abschnitte schließen mit dem Nachweis der korrekten Funktionalität sowie einer Auswertung der erzielten Ergebnisse.
Im weiteren Verlauf wird zuerst die praktisch realisierte Toolchain vorgestellt.

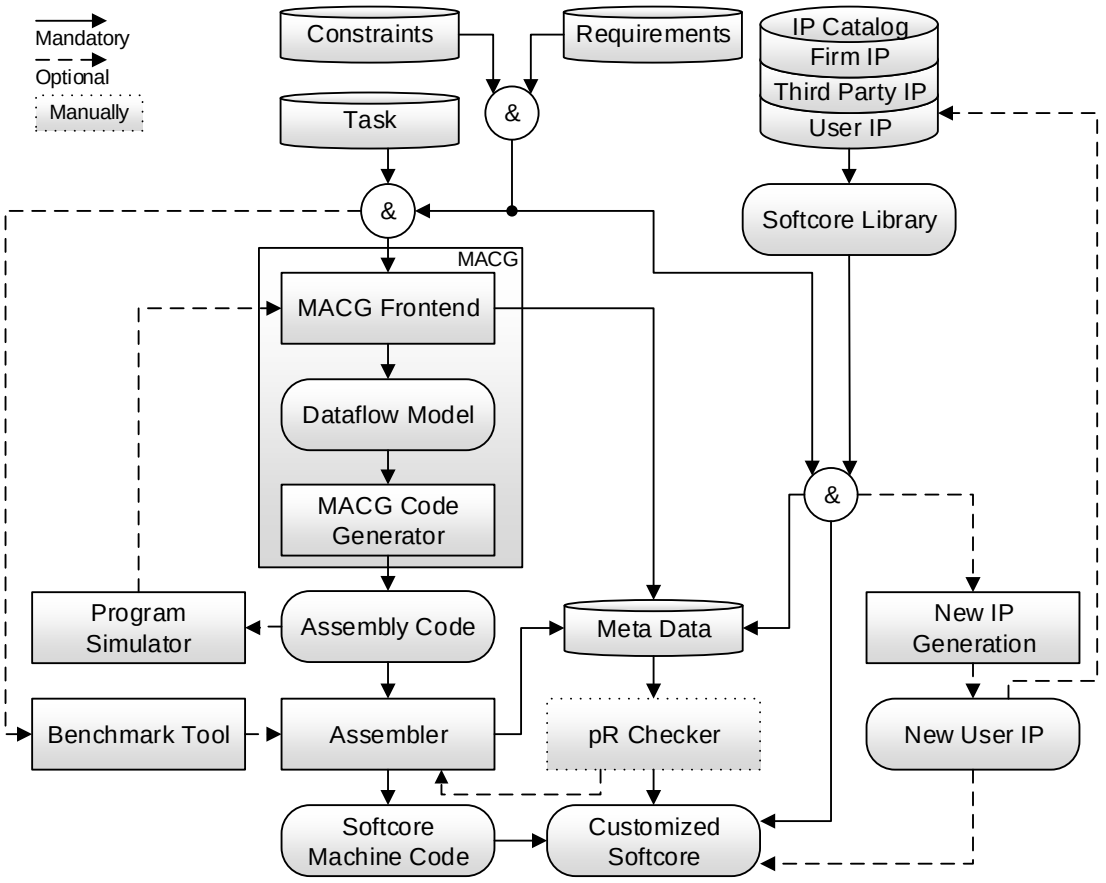



Abbildung 5.1: Praktische Realisierung der Toolchain

Die Toolchain wurde, wie in Abschnitt 4.3 vorgestellt, umgesetzt. Zusätzlich wurden zwei weitere Tools konzipiert und realisiert.

Wie in Abbildung 5.1 zu sehen ist, wurde eine Softcore-Bibliothek (*Softcore Library*) erstellt. Diese Softcore-Bibliothek legt die Rahmenbedingungen, wie den Aufbau des Maschinencodes oder die Definition der verfügbaren Operatoren, fest. Sollte es projektbedingt notwendig sein, ist es möglich dem Softcore weitere Logik hinzuzufügen (*New IP Generation*), was ggf. eine Anpassung des Assemblerbefehlssatzes und der Modelbased Assembly Code Generator (MACG)-Bibliothek nach sich zieht. Beide Tools sind so erstellt, das eine Anpassung und Erweiterung vorgesehen ist und problemlos durchgeführt werden kann.
Der MACG ist ein in Matlab/Simulink umgesetztes eigenständiges Plugin, mit dessen Hilfe modellorientiert Datenflussdiagramme erstellt werden können. Dieses Plugin generiert einen optimierten Assemblercode. Der MACG besteht aus verschiedenen Modulen: zum einen dem Frontend (*MACG Frontend*), mit dessen Hilfe Datenflussdiagramme erstellt werden. Es besteht weiterhin aus dem Code Generator (*MACG Code Generator*) der die Datenflussdiagramme in funktionsäquivalente, sequentialisierte und optimierte Assemblercodes übersetzt. Diese Assemblercodes werden von einem Assembler in für den Softcore lesbaren Maschinencode übersetzt wird. Dieser führt bei der Übersetzung ein pipelineoptimierendes Scheduling aus. Der Softcore (*Customized Softcore*) wird mit Hilfe der Metainformationen (*Meta Data*) auf das Problem angepasst. Eine automatisierte Überprüfung, ob eine partielle Rekonfiguration problemspezifisch sinnvoll ist, wurde nicht umgesetzt. Diese Überprüfung (*pR Checker*) muss manuell erfolgen. Generell wurden partielle Rekonfigurationsmöglichkeiten allerdings untersucht sowie umgesetzt und werden in dem entsprechenden Abschnitt im Detail vorgestellt.
Weiterhin wurden der praktischen Realisierung zwei weitere Tools, ein Programmsimulator (*Program Simulator*) sowie ein Benchmark Tool (*Benchmark Tool*) hinzugefügt. Der Programmsimulator dient dem Testen des vom MACG erzeugten Assemblercodes (*Assembly Code*), was sowohl zum Debugging für neue oder geänderte Features des MACG selbst, als auch zum Testen der erstellten Assemblerprogramme verwendet werden kann.
Das Benchmark Tool wird genauer in Abschnitt 5.3.1 vorgestellt und dient der Einstellung und Anpassung der Optimierungsalgorithmen des umgesetzten Assemblers sowie der quantitativen und qualitativen Analyse der Optimierungsalgorithmen.

Der Softcore Prozessor trägt den Namen „ViSARD“, was eine Abkürzung für VHDL Integrated Softcore Architecture for Reconfigurable Devices ist und wird im Folgenden Abschnitt vorgestellt. Der ViSARD wird aktuell in den im Vorwort vorgestellten Projekten WLI SoC und EKOM eingesetzt. Im Rahmen dieser Projekte führt der Softcore Echtzeit 3D-Profilmessungen von Testobjekten im Nanometerbereich durch.

## 5.1 Der ViSARD Softcore Prozessor

Der ViSARD (*VHDL Integrated Softcore Architecture for Reconfigurable Devices*) Softcore Prozessor ist ein hart echtzeitfähiger applikationsspezifischer Softcore Prozessor, der als generisch anpassbare Softcore-Bibliothek vorliegt. Der Prozessor kann aktuell sowohl im Single-Precision (32 Bit) oder im Double-Precision (64 Bit) Modus betrieben werden. Die intern verwendete Zahlendarstellung richtet sich dabei nach dem IEEE-Gleitkomma Format [IEE85] um maximale Genauigkeit bei allen intern durchgeführten Berechnungen zu garantieren. Die Eingabe oder Ausgabe von Werten kann allerdings in beliebigen Zahlenformaten erfolgen.
Der Funktionsumfang beläuft sich aktuell auf 33 mögliche Befehle, die in den Tabellen B.1 und B.2 dargestellt werden. Der Softcore wurde dabei so konzipiert, dass die ALU voll modular ist

und entsprechend jederzeit eine Erweiterung oder ein Austausch der Hardwareoperatoren und damit auch des Funktionsumfangs vorgenommen werden kann.

Der Aufbau und die Eigenschaften des verwendeten Assemblercodes werden dabei ausführlich in Anhang B erläutert.
Der ViSARD verfügt über eine fünfstufige Befehlspipeline, wie in Abbildung 5.2 dargestellt. Zusätzlich verfügt jeder Operator innerhalb der ALU intern über eine Pipeline.


Abbildung 5.2: Fünfstufige Befehlspipeline

Damit kann in jedem Takt eine Operation gestartet werden, was eine sehr schnelle und effiziente Abarbeitung von Programmcode ermöglicht.
Der ViSARD verfügt über eine Hardwareschleife zur Nachbildung von Schleifenkonstrukten, die im zugrunde liegenden Assemblercode nicht vorgesehen sind, da eine taktgenaue und deterministische Vorhersage der Rechenzeit des Softcores auf diesem Wege sichergestellt wird. Diese ermöglicht es, nach der Abarbeitung eines zur Designzeit festgelegten Teils des Maschinencodes, an eine ebenfalls zur Designzeit festgelegte Stelle dieses Maschinencodes zu springen. Dieser Vorgang kann mit einer zur Designzeit definierten Anzahl an Iterationen wiederholt werden. So ist es möglich eine Schleife zu realisieren, ohne diese aufwendig im Assemblercode und damit ebenfalls im resultierenden Maschinencode, ausrollen zu müssen. Sprungbefehle sind aufgrund der zur Designzeit notwendigen taktgenauen Analysierbarkeit nicht verwendbar, wie bereits in Abschnitt 4.4.1 und in Anhang B im Detail ausgeführt. Abbildung 5.3 zeigt die schematische Darstellung des Softcores.

In Abbildung 5.3 zu sehen sind der Datenpfad (*Data Path*), in grün dargestellt, und das Steuerwerk (*Control Path*), in braun dargestellt. Das Steuerwerk kontrolliert zu jedem Zeitpunkt den aktuellen Stand in der Abarbeitung des gegebenen Programmcodes über den Programmzähler (*Program Counter*) in Kombination mit einem RAM-Modul (*DPRAM (Program)*). In diesem RAM-Modul sind alle Maschinencodebefehle sequentiell abgelegt, die der Prozessor ausführen soll. Der Programmzähler (*Program Counter*) gibt in jedem Takt die aktuelle Programmadresse an den Programmspeicher (*DPRAM (Program)*) weiter. Dort wird der nächste Maschinenbefehl gelesen und an den Dekodierer (*Decoder*) weitergegeben. Dieser Schritt entspricht dem Befehlscode Laden (*Instruction Fetch*) aus Abbildung 5.2. Der Dekodierer dekodiert den Befehl (*Instruction Decode*, siehe Abbildung 5.2) und gibt alle notwendigen Adressen (*Adresses*) und Steuerflussignale (*Control Signal*) im richtigen Takt an die jeweiligen Module weiter.
Das Laden der notwendigen Argumente, also der Inhalte der Speicherzellen, für die Operationen und die Weitergabe der Daten an die ALU erfolgt im nächsten Schritt (*Operand Fetch*, siehe Abbildung 5.2). Die ALU führt die Operationen aus (*Execute*, siehe Abbildung 5.2) und das Ergebnis wird im lokalen und/oder gemeinsamen Speicher (*Write Back*, siehe Abbildung 5.2) abgelegt.
Eine Besonderheit in dieser Abarbeitung bildet dabei das Setzen-/Rücksetzen-Modul

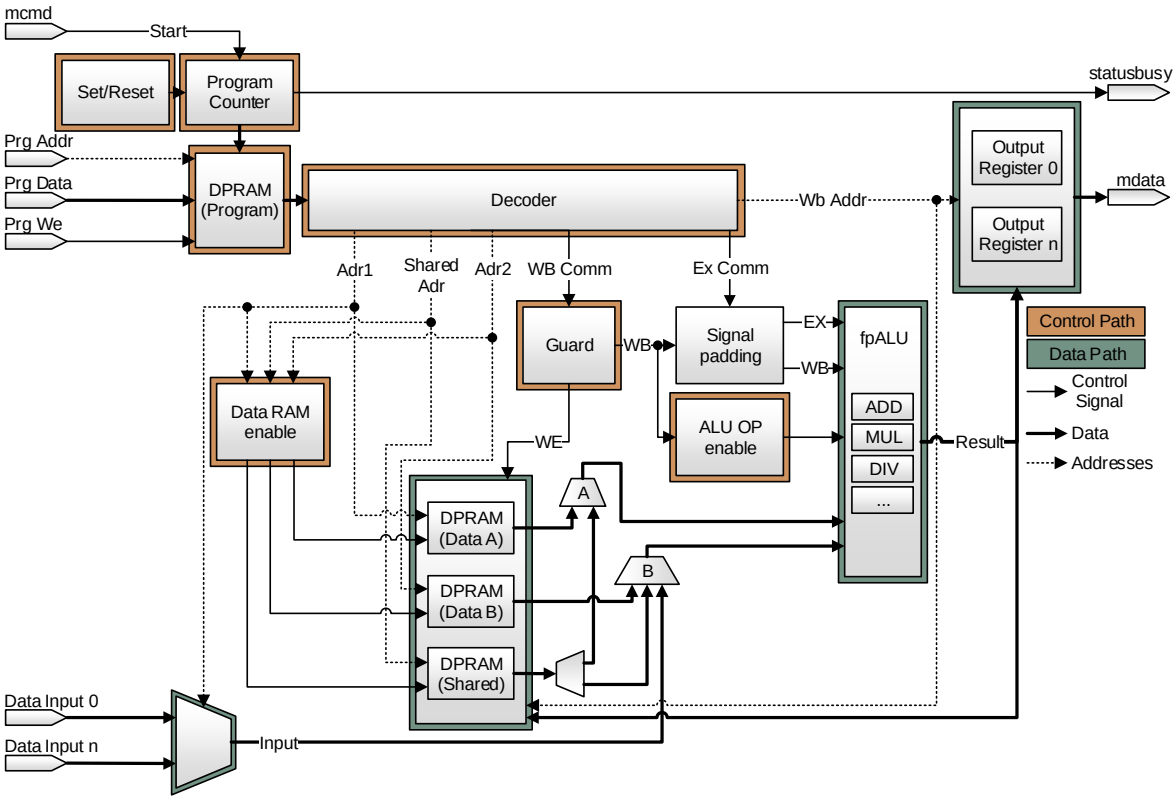



Abbildung 5.3: Schematic des ViSARD Multi-Softcore Prozessor

(*Set/Reset*). Der Programmzähler kann mit diesem Modul in seinem Zählerstand manipuliert werden. Das Modul realisiert die zuvor diskutierte Hardwareschleife. Weiterhin befindet sich im Steuerwerk ein Speicher-Aktivierungsmodul (*Data RAM enable*). Die Aufgabe dieses Moduls ist es, mit Hilfe der Informationen aus dem Dekodierer, den Takt der jeweiligen Datenspeicher abzuschalten, sofern weder eine lesende noch schreibende Operation darauf ausgeführt wird. Mit diesem Mechanismus kann der Energieverbrauch verringert werden.
Ein weiteres Modul, dass dazu dient den Energieverbrauch zu minimieren ist das *ALU OP enable*-Modul. Dieses Modul hat Informationen darüber, in welcher Operatorpipeline sich aktuell Werte befinden die berechnet werden. Sollte eine Pipeline eines Operators leer sein, wird das Modul diesen Operator abschalten, was zu einer weiteren Reduktion des Stromverbrauchs führt.

Das Kernstück des Datenpfades bildet die ALU (*fpALU*). Die Abkürzung fpALU steht dabei für „*floating-point* ALU“. Die ALU (*fpALU*) ist so aufgebaut, dass alle innerhalb dieser verwendeten Operatoren parallel an die Datenverbindungen angeschlossen sind. Auf diese Weise rechnet jeder Operator in jedem Takt mit den anliegenden Argumenten seine jeweilige Operation. Ein Multiplexer innerhalb der ALU entscheidet zu welchem Zeitpunkt welche Ergebnisse von welchen Operatoren abgegriffen und im Speicher abgelegt werden. Als Speicher sind zwei lokale Speicher (*Data A* und *Data B*) sowie ein optionaler gemeinsamer Speicher (*Shared*) zu sehen. Der Speicher ist dabei verflacht, das bedeutet, es wird nur eine Speicherebene verwendet und auf andere Speicherebenen wie Caches, Arbeitsspeicher oder Festplatten wird bewusst verzichtet. Speicher und Register sind also identisch, das bedeutet

jede Speicherzelle entspricht einem Register. Der Grund liegt darin, dass jeder Speicher in jedem Takt sowohl gelesen als auch geschrieben werden muss. Aus diesem Grund werden Zweitorspeicher (Dual-Port RAMs) verwendet, wobei ein Port immer exklusiv zum Schreiben und der zweite Port immer exklusiv zum Lesen verwendet wird. Der Speicher wird doppelt in zwei Speicher-Blöcken (*Data A* und *Data B*) vorgehalten. Diese sind zu jedem Zeitpunkt identisch. Dadruch wird es dem Softcore ermöglicht, in jedem Takt zwei Werte zu lesen (und einen weiteren Wert zu schreiben). Mit dieser Architektur ist garantiert, dass in jedem Takt jeder beliebige Operator bis zu zwei Werte aus den Speicherzellen als Eingabe erhält und darauf die Berechnungen durchführen und ein weiterer Operator sein Ergebnis im Speicher ablegen kann. Da jeder verwendete Operator maximal zwei Argumente als Eingabe erhalten kann, kann also in jedem Takt eine neue Operation gestartet werden. Innerhalb des Datenpfades können externe Eingaben über entsprechende Data Input Ports in den Speicher geschrieben werden. Da diese über einen Assemblerbefehl angesprochen werden müssen, verläuft der Datenpfad über die ALU (*fpALU*).
Der gemeinsame Speicher (*Shared*) wird nur im Falle einer Multi-Softcore Konfiguration des ViSARD verwendet und ist im Falle einer Verwendung nicht mit den prozessorinternen Speicher identisch. Bei jeder Operation kann bis zu ein beliebieges Argument als Eingabe für eine Operation durch den Wert einer Speicherzelle aus dem gemeinsamen Speicher ersetzt werden. Aufgrund der Architektur kann maximal ein Argument je Takt ersetzt werden. Sollte keines der zwei benötigten Argumente im prozessorinternen Speicher verfügbar sein, so wird in einem (zur Designzeit) vorgesetzten Befehl der Inhalt einer Speicherzelle aus dem gemeinsamen Speicher in den prozessorinternen Speicher geschrieben und im Anschluss die eigentliche Operation gestartet.
Das letzte Modul des Datenpfades ist das Ausgabemodul. Dieses beinhaltet eine zur Designzeit definierte und Anzahl an Ausgaberegistern (*Output Register*) und dient dazu entsprechende Ergebnisse des Softcores an die dem Softcore umgebende Logik weiter zu geben. Die Ausgaberegister können über einen Assemblerbefehl unabhängig voneinander angesprochen und belegt werden. Alle Ausgaberegister werden zu einem Datenwort kombiniert und damit als ein kombiniertes Datenwort an die dem Softcore umgebende Logik weitergegeben.

### 5.1.1 Multi-Core Konfiguration

In Abschnitt 4.4.3 wurden verschiedene Konzepte zur Erweiterung eines Softcore Prozessors auf eine Multi-Core-Architektur vorgestellt. Innerhalb des ViSARD wurde das Konzept eines gemeinsamen Speichers, im Rahmen der Masterarbeit von Herrn Wagner [Wag17], umgesetzt. Der Vorteil eines geteilten Speichers im Vergleich zu einer nachrichtenbasierten Synchronisation liegt in der Auslagerung der dafür notwendigen Logik in den Assembler. Dadurch lassen sich innerhalb des FPGA wichtige Ressourcen einsparen, da hier keine Ressourcen für eine Synchronisationslogik benötigt werden. Das liegt in der Tatsache begründet, dass die Synchronisation des Speichers durch den Assembler zur Designzeit erfolgt und etwaige Konflikte im Speicherzugriff durch mehrere Kerne bereits an dieser Stelle behandelt werden.
Da davon auszugehen ist, dass im Einsatzgebiet des Softcore hauptsächlich Multi-Core Konfigurationen mit wenigen Kernen zum Einsatz kommen, wurde eine speicherbasierte sternförmige Verbindungstopologie umgesetzt, wie in Abbildung 5.4 dargestellt ist. Diese bildet nach den aus Abschnitt 4.4.3 unter Verbindungstopologien diskutierten Varianten die besten Eigenschaften im Einsatz innerhalb des ViSARD.

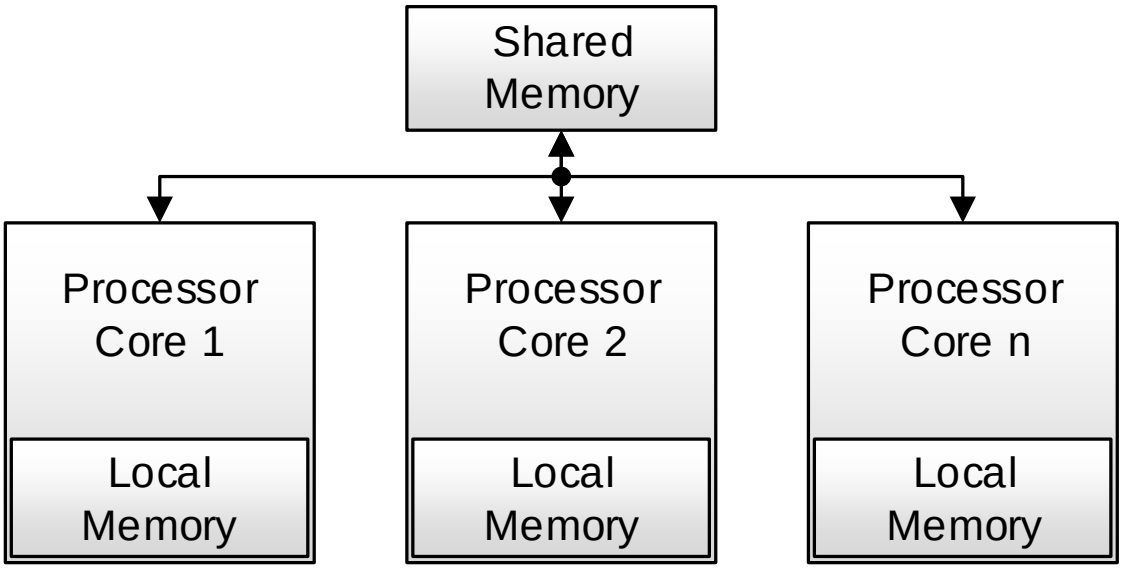



Abbildung 5.4: Speicherarchitektur des Multicore ViSARD

Wie zu sehen ist, kann diese Topologie theoretisch auf beliebig viele Kerne erweitert werden. Wie bereits erwähnt, ist die Architektur darauf ausgelegt Multi-Core Varianten mit wenigen Kernen umzusetzen. Hier werden die Vorteile deutlich: Der Overhead an zusätzlich benötigten Ressourcen innerhalb des FPGA ist minimal, da lediglich ein zusätzlicher Variablenspeicher und eine Datenleitung von jedem Kern zu diesem Speicher benötigt wird. Wann welcher Kern wie auf diesen Speicher zugreift, wird über den Assembler zur Designzeit festgelegt. Zusätzliche Softcore-Module, die dieses Problem und alle dadurch entstehenden Probleme, wie z. B. eine notwendige Deadlock-Behandlung (siehe Abschnitt 5.3.3 unter Abbildung 5.25), lösen müssen, entfallen.

#### Experimente zur Reduktion der Rechenzeit bei mehreren Kernen

Um den Vorteil der Verwendung mehrerer Kerne innerhalb des ViSARD mit der vorgestellten Architektur zu verdeutlichen, werden im folgenden Experiment verschiedene Algorithmen umgesetzt und für verschiedene Konfigurationen des ViSARD die resultierende Rechenzeit angegeben. Dabei wurde die Testkonfiguration nach Anhang D.3 für die Experimente verwendet. Der ViSARD wurde dabei im Rahmen dieser Experimente in den Konfigurationen Single-Core, Dual-Core, Triple-Core und Quad-Core jeweils mit doppelter Genauigkeit (64 Bit Double Precision) getestet. Die folgenden im Detail ausgeführten Algorithmen wurden dabei umgesetzt und verwendet:

- FIR-Filter
  Es wurde ein einfacher FIR-Filter mit einem Eingang und einem Ausgang umgesetzt.
- Kalman-Filter
  Zwei modifizierte Kalman-Filter mit vier bzw. acht Eingängen zur Anwendung als Zustandsschätzer in einem nichttrivialen Regler. Innerhalb des Filters kommen dabei schwachbesetzte Matrizen strukturoptimierte Rechenoperationen zum Einsatz. [PKMF11]

- Positionsregler
  Einen Positionsregler mit drei Achsen (im folgenden Dreiachs Regler genannt), sowie einen Positionsregler mit sechs Achsen (im folgenden Sechsachs Regler genannt) für ein Bewegungssystem mit zwei bzw. drei Translationsachsen und einer bzw. drei Rotationsachsen. Diese Regler enthalten neben PID-Reglern auch Kalman-Filter und weitere Elemente. [DPZ+13, Amt10]

- Polynomberechnungen
  Insgesamt vier Algorithmen zur Berechnung von Summen aus Polynomen. Hierbei wird jeweils die Summe aus zwölf Polynomen dritten Grades sowie die Summe von zwölf, 24 und 48 Polynomen jeweils sechsten Grades errechnet. [DPF12]

- Ellipsenkorrektur
  Ein Algorithmus zur Korrektur von Interferometermesswerten, bestehend aus rekursiver Regression mit vier Parametern, Tiefpass und trigonometrischen Funktionen zur Parameterrückrechnung. Umgesetzt wurde eine Version mit einem kompletten Korrekturdurchlauf und eine weitere Version mit zwei Durchläufen. [HPG+11]

Diese Algorithmen wurden mit Hilfe des Modelbased Assembly Code Generator (*MACG*) umgesetzt, der in Abschnitt 5.2 im Detail vorgestellt wird.
Die folgende Tabelle 5.1 zeigt die resultierende notwendige Rechenzeit der jeweiligen Konfigurationen.

| | Rechenzeit in ns | | | |
|---|---|---|---|---|
| Name | Single-Core | Dual-Core | Triple-Core | Quad-Core |
| FIR | 1.520 | 1.190 | 1.090 | 1.030 |
| Kalman 4 | 2.850 | 2.550 | 2.650 | 2.690 |
| Kalman 8 | 4.370 | 3.060 | 3.580 | 3.900 |
| Dreiachs Regler | 2.160 | 2.010 | 2.370 | 2.540 |
| Sechsachs Regler | 9.910 | 8.340 | 8.760 | 10.820 |
| Polynom 12-3 | 1.860 | 1.780 | 1.800 | 1.850 |
| Polynom 12-6 | 2.350 | 2.950 | 1.920 | 1.900 |
| Polynom 24-6 | 3.590 | 2.550 | 2.340 | 2.930 |
| Polynom 48-6 | 6.260 | 5.240 | 3.950 | 3.730 |
| Ellipse 1x | 15.020 | 15.800 | 15.740 | 16.100 |
| Ellipse 2x | 17.560 | 23.290 | 27.390 | 30.940 |

Tabelle 5.1: Resultierende Rechenzeit der Multi-Core ViSARD Konfigurationen

Wie der Tabelle 5.1 entnommen werden kann, ist die Reduktion der Rechenzeit bei verschiedenen Multi-Core Konfigurationen stark vom jeweiligen Problem abhängig. Bei einigen Algorithmen, wie z. B. dem Polynom 48-6 oder dem FIR-Filter kann mit jedem weiteren Kern im Experiment eine Verringerung der notwendigen Rechenzeit erreicht werden. Andere Algorithmen haben Eigenschaften, die eine effizientere Verarbeitung durch einen Softcore mit mehreren Kernen verhindert. Beispielsweise bei dem Kalman-Filter (Kalman 4) wird zwar bei dem Sprung von einem Single-Core auf den Dual-Core eine Rechenzeitverringerung von 300 ns erreicht. Allerdings ist der Kommunikationsaufwand der einzelnen Kerne untereinander derart groß, dass bereits bei der Triple-Core Version die Rechenzeit dadurch im Vergleich zur Dual-Core Version um 100 ns verlängert wird.

Die praktische Verwendung einer speziellen Multi-Core Version des ViSARD ist also stark problemabhängig und muss daher von Projekt zu Projekt getestet und entschieden werden.

### 5.1.2 Das torCombitgen Tool

In diesem Abschnitt wird auf ein selbst erstelltes Tool eingegangen, mit dessen Hilfe Bitstreams zur partiellen Rekonfiguration erzeugt werden können. Dieses Tool ist nicht Teil der eigentlichen unter Abbildung 5.1 vorgestellten Toolchain.
Die Rekonfigurationszeiten sind maßgeblich von der Größe der übertragenen Bitstreams abhängig. Die von Xilinx erzeugten Bitstreams rekonfigurieren das gesamte zu rekonfigurierende Areal. Das ist allerdings in vielen Fällen nicht notwendig, da Teile der dortigen Ressourcen (LUTs,Flip-Flops,...) nicht neu konfiguriert werden müssen. Diese Eigenschaft nutzt torCombitgen aus. Mit Hilfe dieses Tools können die Bitstreams zur Rekonfiguration gleicher Logik verkleinert und die Rekonfigurationszeit damit minimiert werden. Da gerade bei der partiellen Rekonfiguration von echtzeitkritischen Softcore Prozessoren diese Rekonfigurationszeit ein wichtiges Kriterium ist, wird das Tool an dieser Stelle vorgestellt.
Das Tool torCombitgen ist ein Tool zur Erzeugung von partiellen Bitstreams für Xilinx FPGAs und basiert auf den Ideen des Combitgen-Tools [CMS06] und ist eine im Rahmen dieser Arbeit vorgenommene Anpassung und Erweiterung dieses Tools an moderne FPGA-Familien. Da Combitgen lediglich für mittlerweile sehr alte FPGA-Familien, wie beispielsweise Virtex II, konzipiert wurde, musste ein neues Tool realisiert werden, dass die sich ändernden Eigenschaften und Aufbauten neuerer FPGA-Familien berücksichtigt.
Mit Hilfe dieses Tools können partielle Bitstreams für alle Xilinx-FPGAs der Familien der 7-Series und zusätzlich alle Virtex-FPGAs von Virtex II bis zur 7-Series erzeugt werden. Das torCombitgen Tool stellt damit eine Anpassung des aus der Literatur vorgefundenen Combitgen an neue FPGA-Familien dar.
Das torCombitgen liest verschiedene Bitstreams und identifiziert identische Frames und löscht diese im neu erzeugten und dadurch minimierten Bitstream.

Das Tool benutzt dabei die Open Source Bibliothek TORC (*Tools for Open Reconfigurable Computing*). [SWS+11]
Es werden minimal zwei Toplevel Bitstreams (also komplette, keine partiellen Bitstreams) dem Tool als Eingabe übergeben. Die Toplevel Bitstreams realisieren verschiedene Implementierungen von rekonfigurierbaren Modulen. Diese werden eingelesen und die jeweilig kleinstmöglichen adressierbaren Größen, also einzelne Konfigurationsframes, werden miteinander verglichen. Dabei werden alle Konfigurationsframes markiert, bei denen ein Unterschied festgestellt wird. In den resultierenden Bitstream werden im Anschluss alle markierten Konfigurationsframes geschrieben.
Der Vorteil der durch torCombitgen erstellten partiellen Bitstreams, im Vergleich mit den Xilinx-erstellten partiellen Bitstreams, liegt darin, dass in den Hersteller-Bitstreams immer die gesamte zu rekonfigurierende Region neu konfiguriert werden muss. Innerhalb entsprechender Regionen gibt es allerdings praktisch immer Ressourcen, die für die jeweilige Konfiguration nicht benötigt werden. Diese Ressourcen müssen nicht mit konfiguriert werden und so ist es möglich die Größe des resultierenden partiellen Bitstreams zu minimieren. Da die Rekonfigurationszeit immer von der Größe des partiellen Bitstreams abhängig ist, kann diese damit reduziert werden.

Wie in Abbildung 5.5 gesehen werden kann, werden die Tolevel Bitstreams auf einen torCombitgen internen Pseudo-FPGA-Speicher (*Pseudo-FPGA-Memory*) geladen. Dabei werden

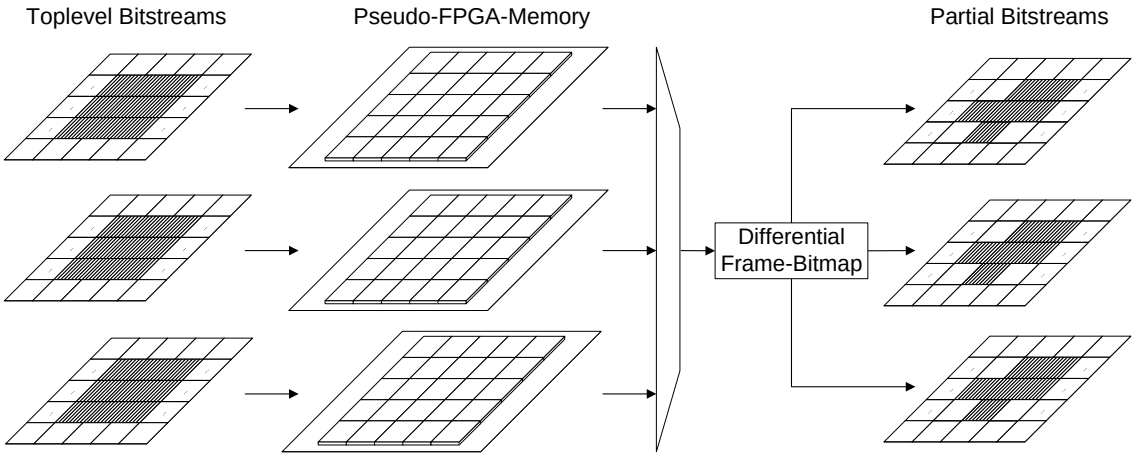


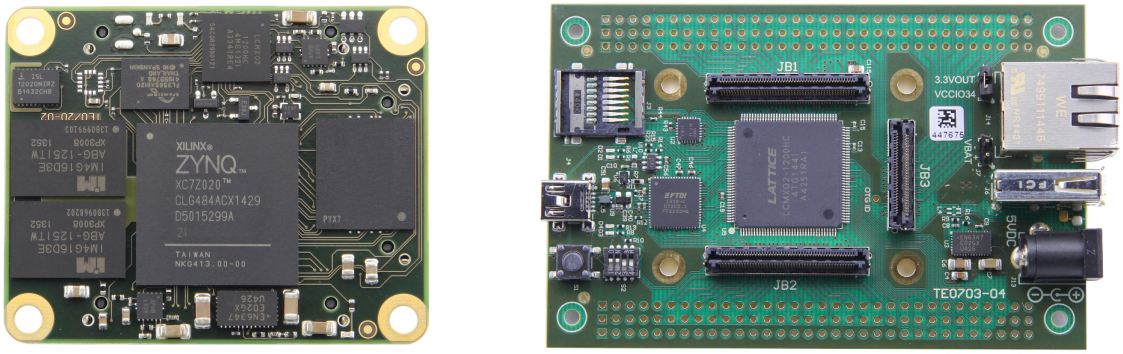

Abbildung 5.5: Funktionsweise des torCombitgen (angelehnt an [CMS06])

alle Konfigurationsframes miteinander verglichen und unterschiedliche Frames markiert. Alle markierten Konfigurationsframes werden im Anschluss in einer Bitmap (*Differential Frame-Bitmap*) gespeichert. In einem letzten Schritt werden aus diesen Informationen die minimierten jeweiligen partiellen Bitströme (*Partial Bitstreams*) erzeugt.

### 5.1.3 Umsetzung des partiell rekonfigurierbaren ViSARD

Die Entwicklung von Softcore Prozessoren nach dem aus Abschnitt 4.1 vorgestellten Φ-Vorgehensmodell ist dazu gedacht, allgemeingültig für alle Softcore Prozessoren verwendet zu werden. Der ViSARD Softcore ist nach diesem Vorgehensmodell entwickelt worden und entsprechend in seiner Grundform nicht für einen speziellen FPGA entwickelt. Dieser Softcore läuft auf allen FPGAs der Firma Xilinx, sofern diese genügend Ressourcen besitzen um mindestens die Single-Core Version des ViSARD zu realisieren. Der Einsatz in der in Abschnitt 2.6 vorgestellten Aufgabendomäne und die daraus resultierende Eigenschaft der sich über die Verarbeitungszeit ändernden Anforderungen eröffnen die Möglichkeit, das Prozessordesign während der Laufzeit partiell zu rekonfigurieren. In der in Abbildung 4.3 vorgestellten und in Abbildung 5.1 praktisch umgesetzten Methodologie der Toolchain spielt die partielle Rekonfiguration eine wichtige Rolle. Deswegen ist es notwendig entsprechende Untersuchungen zur partiellen Rekonfiguration durchzuführen.
Diese partielle Rekonfiguration bringt, je nach Einsatzszenario, verschiedene Vorteile wie beispielsweise eine Verringerung der benötigten Chipfläche oder eine Erhöhung der Verarbeitungsgeschwindigkeit einzelner Operatoren innerhalb des Prozessors bei gleichbleibendem Ressourcenbedarf. Ein Nachteil ist die Einschränkung auf eine festgelegte FPGA-Familie, da spezielle zur partiellen Rekonfiguration benötigte Teile FPGA-spezifisch sind.
Ein genereller Aufbau eines solchen Softcore Prozessors wurde in Abschnitt 3.2 unter Abbildung 3.5 einem Aufbau ohne partielle Rekonfigurierbarkeit gegenübergestellt. Dieser rekonfigurierbare Aufbau ist allerdings nur ein sehr vereinfachter Aufbau und muss für den praktischen Einsatz verfeinert werden.
Wie bereits in Abschnitt 4.4.4 erläutert wurde, wird für eine partielle Rekonfiguration ohne externe Unterstützung ein partieller Rekonfigurationscontroller (*PRCO*), eine Steuerlogik (*Control Logic*) sowie eine Speicherlösung für die partiellen Bitströme benötigt. Die Notwendigkeit dieser

Komponenten ergibt sich aus einer Analyse (*Evaluation*, siehe Abbildung 4.1 des Φ-Modells), der vorhandenen Umsetzung des ViSARDs (*Softcore Implementation*) und ergab eine entsprechende Erweiterung der ViSARD Architektur (*System Architecture*) um die Rekonfigurationslogik.
Der PRCO bildet dabei den Kern der Rekonfigurationslogik. Der PRCO ist in Anhang C.2 als IP-Block mit detaillierten Angaben über die Portanbindung zu finden. Im Folgenden werden alle Anpassungen an den ViSARD zur partiellen Rekonfiguration sowie die dafür zusätzlich realisierte Logik vorgestellt.
Wie bereits zuvor erläutert, ist es notwendig Informationen über die verwendete Hardware in das Design und gewisse Designentscheidungen mit einfließen zu lassen. Aus diesem Grund wird zunächst die für diesen Teil verwendete Hardware mit den relevanten Eigenschaften kurz besprochen.




(a) TE0720 (Quelle: [Tre18b]) (b) TE0703 Trägerboard (Quelle: [Tre18a])

Abbildung 5.6: Zur Realisierung verwendetes FPGA Board

Verwendet wird das TE-0720 GigaZee System-on-Module (*SOM*) der Firma Trenz Electronic, das in Abbildung 5.6a gezeigt ist, in Kombination mit dem Trägerboard TE-0703, das in Abbildung 5.6b zu sehen ist. Das FPGA Modul ist dabei mit einem Xilinx Zynq Z-7020 FPGA ausgestattet, verfügt über einen Gigabit Transceiver, 32 MB Quad Serial Peripheral Interface (*QSPI*)-Flash Speicher, 4GB embedded Multi Media Card (*eMMC*)-Speicher und 1GB DDR3 SDRAM. Der intern verwendete FPGA verfügt unter anderem über 0,6125 MB BRAM. [Tre17] Die Informationen über die verschiedenen aufgelisteten Speichertypen samt Kapazitäten werden im weiteren Verlauf des Abschnitts eine entscheidende Rolle spielen, wenn eine Speicherlösung für den PRCO ausgewählt und realisiert wird.

Der Zynq bietet zwei CAP-Ports: den Prozessorkonfigurationszugriffsport (*Processor Configuration Access Port* (*PCAP*)) und den internen Konfigurationszugriffsport (*Internal Configuration Access Port* (*ICAP*)) zur (re-)Konfigurierung des FPGAs. Der PCAP kann dabei von dem umgebenden System direkt als (re-)Konfigurationsport genutzt werden, während der ICAP lediglich zur Rekonfiguration innerhalb des FPGAs genutzt werden kann. Die jeweiligen Ports können dabei nur exklusiv, also nicht gleichzeitig, verwendet werden. Der PCAP ist nur auf dem Zynq verfügbar und wird aus diesem Grund nicht zur Verwendung in Betracht gezogen. Auch wenn bei der partiellen Rekonfiguration Teile des Designs FPGA-spezifisch sind, so soll einerseits darauf geachtet werden, dass möglichst viele Teile des Designs mit anderen FPGAs kompatibel bleiben. Andererseits sollen die Experimente möglichst aussagekräftig auch bei der partiellen Rekonfiguration auf anderen FPGAs sein. Bei der Verwendung des

ICAP-Ports, der auch in anderen FPGA-Familien des gleichen Herstellers mit meist sogar den exakt gleichen Spezifikationen verwendet wird, ist die Aussage der Tests auf andere FPGA-Familien übertragbar. Ein zusätzliches Problem des PCAP-Ports liegt in einem maximalen Durchsatz von 145 MB/s, verglichen mit dem maximalen Durchsatz des ICAP von 400 MB/s auf dem Zynq. Praktische Tests zeigen eine reale Geschwindigkeit des PCAP-Ports von ca. 128 MB/s. [Xil18g, VF14, RA14] Nach der Anforderung, die schnellstmögliche Realisierung zur partiellen Rekonfiguration zu verwenden und zur besseren Aussagekraft der Ergebnisse der Tests, entfällt der PCAP-Port als möglicher Realisierungsport.
Um Zugriff auf den ICAP-Port zu erhalten, ist es notwendig das ein PRCO verwendet wird. Hierbei gibt es verschiedene Möglichkeiten zur Realisierung des PRCOs: Die einfachste Realisierung verwendet einen IP-Core, den „AXI Hardware Internal Configuration Access Port IP Core“ [Xil16b], der lediglich einen Zugangspunkt des ICAP-Ports über einen AXI-Bus darstellt. Die gesamte Kontrollogik wird im Anschluss in Software realisiert.
Die Firma Xilinx bietet weiterhin einen speziellen partiellen Rekonfigurationscontroller [Xil18b] an, der über ein einfaches Triggersignal eine zuvor spezifizierte Konfiguration in die festgelegte zu rekonfigurierende Region lädt. Dabei werden bis zu 32 Rekonfigurationsregionen mit bis zu 128 verschiedenen Konfigurationen unterstützt. Allerdings werden bei diesem IP-Core keine Garantien gegeben, wann eine Rekonfiguration gestartet bzw. beendet wird. Echtzeitkritische Probleme können mit diesem IP-Core also nicht umgesetzt werden.
Es kann ein weiterer Softcore Prozessor, beispielsweise ein Microblaze [Xil17b] in Kombination mit einem AXI-Bus verwendet werden. Der Vorteil bei dieser Realisierung liegt darin, dass der Microblaze ein generischer Softcore ist, der von der Firma Xilinx für praktisch alle FPGA-Familien zur Verfügung gestellt wird und direkt verwendet werden kann. Dieser zusätzliche Softcore würde allerdings einen entsprechenden Overhead im Ressourcenverbrauch bedeuten, weswegen diese Lösung nicht optimal ist.
Eine weitere Möglichkeit ist die Verwendung der auf dem Zynq befindlichen CPU, anstatt einer zusätzlichen Softcore Lösung. Eine beispielhafte Implementierung ist in [VF14] vorgestellt. Diese Realisierung ist Zynq-speziell und kann damit nicht auf andere FPGA-Plattformen portiert werden.
Zuletzt sei erwähnt, dass in der Literatur weitere PRCOs vorgestellt werden, die als reines Hardwaredesign erstellt wurden und damit keinen weiteren Prozessor, sowohl Hard.- als auch Softprozessor, benötigen. Als Beispiele sind hier zu nennen: [TEKV14, BKT14, LD09]. Da diese Designs allerdings nicht zugänglich sind, konnten Sie für diese Arbeit nicht verwendet werden.

Aufgrund der verschiedenen Nachteile bereits existierender Lösungen wurde eine eigenständige und auf den ViSARD speziell angepasste Realisierung eines PRCO umgesetzt. Dabei werden die unter Abschnitt 4.4.4 vorgestellten Eigenschaften umgesetzt. Auf die dort erwähnten optionalen Eigenschaften wird bewusst verzichtet, da diese für den Einsatzzweck innerhalb des ViSARD nicht notwendig sind und lediglich zu einer Erhöhung des Overheads an Ressourcenverbrauch führen würden:

- Bitstream (De-)Kompression
  Eine Kompression und anschließende Dekompression würde zu einer Erhöhung des benötigten Ressourcenoverheads führen, da diese Logik zusätzlich implementiert werden müsste und könnte die Rekonfigurationsgeschwindigkeit verringern.

- Verwendung von Checksummen
  Der verwendete Speicher ist physisch an den PRCO angeschlossen und bietet eine

ausreichende Zuverlässigkeit, sodass davon ausgegangen werden kann, dass keine Bitfehler auftreten. Das Berechnen und Auswerten von Checksummen ist also nicht notwendig.

- Bitstream Kompatibilitätsüberprüfung
  Dieses Sicherheitsfeature ist nicht notwendig, da davon ausgegangen wird, dass nur korrekte und kompatible partielle Bitstreams verwendet werden.

Die wesentliche Aufgabe des PRCO ist es also, die partiellen Bitstreams mit einer Geschwindigkeit aus dem Speicher an den ICAP weiterzugeben, sodass dieser im Zeitraum der Datenübertragung immer voll ausgelastet ist. Weiterhin muss der PRCO die harten Echtzeitkriterien erfüllen. In diesem Zusammenhang wird im Folgenden die gewählte Speicherlösung diskutiert.
Der innerhalb des FPGA verfügbare BRAM ist die schnellste Speicherlösung. Da dieser allerdings im Vergleich zu allen weiteren Optionen ein sehr kleiner Speicher ist, wie anfangs des Abschnitts bereits aufgezeigt, und zusätzlich als Ressource für den ViSARD verwendet wird, kommt dieser nicht zur Speicherung der Bitstreams in Frage. Der verfügbare Flash-Speicher, genauso wie der eMMC-Speicher bieten als alleinige Speicherlösung nicht genügend Geschwindigkeit, um den ICAP (400 MB/s) voll auszulasten: Der QSPI-Flash Speicher kann mit einer maximalen Frequenz von 100 MHz betrieben werden und je Takt 4 Bit ausgeben, was in eine theoretische maximale Transferrate von 50 MB/s realisiert. Weiterhin kann der eMMC-Speicher mit maximal 50 MHz betrieben werden, ebenfalls bei 4 Bit je Takt, was theoretisch maximal 25 MB/s ermöglicht. [Xil18h]
Der auf der Hardware vorhandene DDR3 SDRAM hat eine maximale Bandbreite von 4.264 MB/s. [Xil18g] Um auf diesen zugreifen zu können, muss ein hochperformanter AXI-HP Bus verwendet werden. Dieser hat eine maximale Bandbreite von 1.200 MB/s bei einer Frequenz von 150 MHz, was ausreichend ist um den ICAP voll auszulasten. Zusätzlich ist, bei der Verwendung dieses Speichers, eine maximale Latenz beim Speicherzugriff vom Hersteller angegeben und garantiert damit die Einhaltung der Echtzeitkriterien. Der Nachteil der Verwendung von SDRAM liegt darin, dass dieser die Bitstreams nicht permanent speichern kann. Das bedeutet, dass sämtliche Bitstreams bei einem Neustart des Gerätes erneut auf dem SDRAM gespeichert werden müssen.
Da dieses Speichern der partiellen Bitstreams auf den SDRAM aber nicht zeitkritisch ist und vor dem Start der/des ViSARDs erfolgen kann, kann dies von einem anderen Speicher, auf dem eine permanente Speicherung möglich ist, erfolgen. Aus diesem Grund wurde eine duale Speicherlösung umgesetzt, bei der die permanente Speicherung der partiellen Bitstreams auf dem eMMC-Speicher erfolgt. Die Bitstreams werden dann beim Start automatisch in den SDRAM geladen und sind dort für den PRCO verfügbar, sobald eine partielle Rekonfiguration notwendig wird.

In Abbildung 5.7 wird die angepasste Speicherstruktur, wie zuvor beschrieben, dargestellt. Der ICAP arbeitet bei einer spezifizierten maximalen Frequenz von 100 MHz. Es gibt Ansätze, den ICAP auf einer höheren Frequenz arbeiten zu lassen um die Rekonfigurationsgeschwindigkeit zu erhöhen. Als Beispiel kann [HKT11] genannt werden. In dieser Publikation wird mittels Übertaktung und einer speziell entwickelten dem ICAP umgebenden Logik eine maximale Frequenz von 550 Mhz und damit ein maximaler Durchsatz von 2.200 MB/s erreicht. Ein Übertakten des ICAP wurde allerdings nicht vorgenommen, da der Hersteller auf einer Frequenz höher als maximal spezifiziert keine Garantien über die Richtigkeit der übertragenen Bitstreams gibt. Ein Laden eines falschen Bitstreams würde zu einem unvorhersagbaren Verhalten des

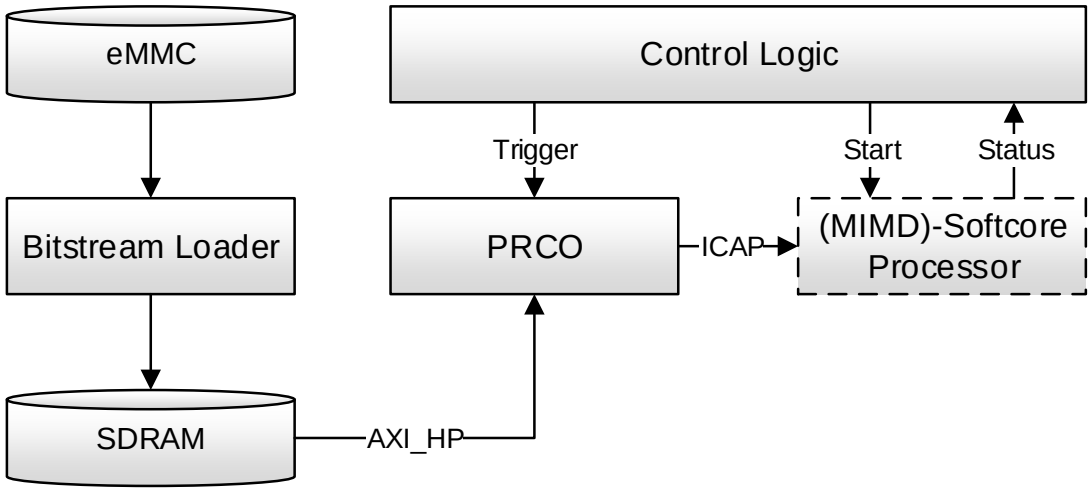



Abbildung 5.7: Struktur der Speicherlösung

Softcore Prozessors führen und damit unter anderem auch das Kriterium nach Vorhersagbarkeit und harter Echtzeit verletzen.
Der ICAP wird also mit einer Frequenz von 100 MHz betrieben und verarbeitet je Takt bis zu 32 Bit. Der AXI-Bus zur Bereitstellung der partiellen Bitströme aus dem SDRAM wird zur Vereinfachung ebenfalls auf 100 MHz festgelegt, da diese Frequenz und die daraus resultierende Übertragungsrate für den ICAP ausreichend sind. Der AXI Bus ist in der Lage in jedem Takt bis zu 64 Bit zu übertragen, was dem doppelten des vom ICAP empfangbaren Datenvolumen entspricht. Da es möglich ist, dass der AXI Bus einige Takte zwischen Anfrage und Lieferung der Daten benötigt, kann dies durch diesen Overhead ausgeglichen werden. Um die Daten entsprechend zu puffern wird eine FiFo mit einem 64 Bit Eingang und einem 32 Bit Ausgang verwendet.
Der PRCO verwendet in der Realisierung ca. 251 LUTs und 294 Flip-Flops (*FFs*), sowie eine BRAM-Einheit für die Puffer-FiFo (diese Werte variieren leicht aufgrund der von der Vivado Design Suite[1] verwendeten nicht deterministischen Algorithmen). Der Overhead an Ressourcen, die zusätzlich für den PRCO benötigt werden, ist damit sehr gering.

### 5.1.4 Experimente zur partiellen Rekonfiguration des ViSARD

Wie bereits in Abschnitt 2.4.2 erläutert, müssen Rekonfigurationsareale festgelegt werden, innerhalb derer sich die jeweils zu rekonfigurierenden Teile des Softcores befinden. Diese Rekonfigurationsareale werden auf Xilinx FPGAs als pBlocks [Xil18e] bezeichnet. Da die partielle Rekonfiguration des ViSARD auf Xilinx FPGAs umgesetzt wurde, wird im Weiteren dieser Begriff verwendet. Wie in Abbildung 2.7 gezeigt, muss jeder pBlock (dort als *Reconfigurable Block* bezeichnet) über genügend Ressourcen verfügen, um alle zu rekonfigurierenden Teile realisieren zu können. Dabei gibt es allerdings weitere Einschränkungen die im Folgenden erläutert werden:

[1] Die Vivado Design Suite ist eine Software-Suite der Firma Xilinx für die Synthese und Analyse von HDL-Designs zur FPGA-Entwicklung.

Wie in Abbildung 2.5 zu erkennen, besteht ein FPGA aus verschiedenen Regionen, die potenziell unterschiedlich getaktet werden können, sogenannte Clock-Regions. Diese Clock-Regions können in der Abbildung als dünne Rahmen um die jeweiligen Teile des FPGAs erkannt werden, z. B. die gelbe Region „X0Y1“. Die oberen und unteren Grenzen der pBlocks sollten immer mit den Grenzen dieser Clock-Regions übereinstimmen. Ist dies nicht der Fall, können zwei Probleme auftreten:

- Statische Logik
  Unter statischer Logik wird Logik verstanden, die sich während der gesamten Zeit ab der initialen Konfiguration auf dem FPGA befindet und sich nicht ändert (im Vergleich zu dynamisch austauschbarer Logik).
  Ein Konfigurationsframe, wie in Abbildung 2.6 vorgestellt, ist die kleinste adressierbare Einheit bei der Rekonfiguration. Stimmen die Grenzen des pBlocks nicht mit den Grenzen der Clock-Region überein, gibt es Konfigurationsframes, in denen sich statische Logik befinden kann, da diese Frames immer mit den Grenzen der Clock-Region enden und in einem entsprechenden Fall mit statischer Logik aufgefüllt werden. Das bedeutet allerdings, dass dieser Teil der statischen Logik ebenfalls überschrieben wird, sofern eine Rekonfiguration stattfindet. Das ist genau dann ein Problem, wenn die statische Logik BRAMs, LUTs oder Shift-Register beinhaltet. Diese Elemente würden ihre Zustände verlieren und damit zu einem unvorhersagbaren Verhalten führen. [Xil18e]

- Bitstreamgröße
  Auch wenn die statische Logik nicht verändert wird, muss diese dennoch, zur gleichen Logik, rekonfiguriert werden, sofern sich diese innerhalb des Konfigurationsframes befindet. Das bedeutet, der jeweilige Bitstream muss die Informationen dafür bereitstellen. Das resultiert in einem unnötig großen Bitstream, was die Rekonfigurationszeit negativ beeinflusst.

Entsprechende pBlocks sollten immer in einer rechteckigen Form definiert werden. Es ist zwar möglich, andere Formen zu wählen, wie beispielsweise eine U-Form oder eine T-Form, das kann allerdings zu Problemen beim Routing der Signale führen. [Xil18e]
Sollten innerhalb eines pBlocks Ressourcen liegen, die für eine Rekonfiguration nicht benötigt werden, wie beispielsweise BRAMs oder DSPs, stehen diese nicht für die statische Logik zur Verfügung. Das ist vor allem dann relevant, wenn pBlocks über entsprechende Ressourcen aufgeschlagen werden, um genügend LUTs oder Flip-Flops zu beinhalten.
Um die korrekte Funktionsweise der vorgestellten Lösung nachzuweisen, wurden verschiedene Konfigurationen mit unterschiedlichen Rekonfigurationsgranularitäten, wie in Abschnitt 4.4.4 vorgestellt, ausgewählt und werden im Weiteren diskutiert. Es ist anzumerken, dass alle im Folgenden vorgestellten Ergebnisse mit dem ViSARD Softcore in doppelter Genauigkeit (64 Bit) erzeugt wurden. Diese sind allerdings repräsentativ und gelten ebenfalls für die einfache Genauigkeit (32 Bit), da die gemessenen Rekonfigurationszeiten in beiden Fällen von den Bitstreamgrößen abhängen.

#### Experimentell überprüfte Bitstreams

In diesem Abschnitt werden die in den Experimenten untersuchten Bitstreams vorgestellt.
Der erste vorgestellte Bitstream ist der durch die Vivado-Suite erzeugte Standardbitstream. Dieser wird in den Experimenten unter der Bezeichnung „Xilinx Standard“ verstanden. Dieser

enthält alle Informationen, die zum Rekonfigurieren eines festgelegten Areals notwendig sind. In diesem Bitstream sind ebenfalls sogenannte „Blanking Events“ [Xil18e] enthalten. Blanking in ein Verfahren, dass die Logik in der zu rekonfigurierende Region komplett löscht, bevor diese erneut Konfiguriert wird. Das bedeutet, mit diesem Bitstream wird die zu rekonfigurierende Region immer exakt zweimal rekonfiguriert.
Ein weiterer untersuchter Bitstream ist der Standardbitstream mit deaktivierten Blanking Events (als „Standard ohne Blanking“ bezeichnet). Der einzige Unterschied besteht darin, dass das zu rekonfigurierende Areal nur einmal rekonfiguriert wird und das Blanking entfällt.
Eine Spezialisierung des Standardbitstreams bildet der komprimierte Bitstream (als „Xilinx Kompression“ bezeichnet). Dieser Bitstream ist nicht komprimiert im klassischen Sinn. Das bedeutet, dass der PRCO diesen Bitstream nicht im Vorfeld der Übertragung über den ICAP komprimiert und der Bitstream im Anschluss nicht Dekomprimiert werden muss. Als „Xilinx Kompression“ wird ein Bitstream verstanden, der Multiple Frame Write-Befehle verwendet, um identische Konfigurationsframes gleichzeitig in mehrere Adressen zu schreiben. Mit dieser Methode ist es möglich, den resultierenden Bitstream zu verkleinern, ohne das Informationen verloren gehen oder eine Dekompression notwendig ist. Dieser Bitstream enthält ebenfalls die Blanking Events. [Xil18a]
Eine weitere Methode, um Bitstreams zu erzeugen, die im Rahmen der Experimente verwendet wurde, ist die differenz-basierte Methode. Hierbei werden, vergleichbar mit dem Ansatz des torCombitgen, exakt zwei Toplevel Bitstreams miteinander verglichen und lediglich die sich unterscheidenden Konfigurationsframes in die resultierenden partiellen Bitstreams geschrieben. Der Nachteil gegenüber dem Ansatz des torCombitgen Tools liegt darin, das immer nur exakt zwei Toplevel Bitstreams miteinander verglichen werden. Daraus resultiert eine sich mit Anzahl der Rekonfigurationen stark erhöhende Anzahl an partiellen Bitstreams, da jeder partielle Bitstream immer nur die Möglichkeit bietet von exakt einer definierten Konfiguration zu exakt einer weiteren Konfiguration zu Rekonfigurieren. Die Anzahl an benötigten partiellen Bitstreams ($B$) richtet sich nach der Gleichung 5.1. [CMS06]
In Gleichung 5.1 steht $n$ für die Anzahl an Designs, also die Anzahl der Toplevel.

$$B = \prod_{k=n-1}^{n} k = n^2 - n \tag{5.1}$$

Zusammen mit dem durch das Tool torCombitgen erzeugte Bitstream wurden also fünf verschiedene Bitstreams in jedem Experiment untersucht, wobei der durch torCombitgen erzeugte Bitstream den einzig nicht durch die Vivado-Suite erstellten Bitstream darstellt.

#### Komponentenrekonfiguration in der ALU

In diesem Experiment wurde die kleinste sinnvolle Rekonfigurationsgranularität umgesetzt: die Rekonfiguration von einzelnen Operatoren (*Execution Units* (*EU*)) innerhalb der ALU. Der Austausch von EUs ist aufgabenabhängig. Es müssen zum Nachweis der korrekten Funtionalität und um die benötigten Rekonfigurationszeiten demonstrieren zu können, zwei Konfigurationen festgelegt werden, die durch die jeweils andere ausgetauscht werden können. Entsprechend wird die Wahl der auszutauschenden EUs möglichst allgemeingültig und problemunabhängig diskutiert.
Eine komplette Liste der vom ViSARD verwendeten EUs werden in den Tabellen B.1 und B.2

im Anhang vorgestellt. Nachfolgend wird eine gezielte Auswahl aus dieser Liste zur partiellen Rekonfiguration getroffen.
Dabei wurde festgestellt, dass alle aktuell verwendeten EUs, die zur Konvertierung verwendet werden, nur vergleichsweise sehr wenige Ressourcen, also z. B. Slices, und keine BRAMs oder DSPs benötigen. Der mögliche Ressourcengewinn bei einer Rekonfiguration zwischen diesen EUs wäre also minimal. Aus diesem Grund werden keine EUs die zu Konvertierungen verwendet werden rekonfiguriert. Das Gleiche gilt für Kopieroperationen wie „Mov“ (*Move*). Auch diese EUs werden nicht rekonfiguriert.
Die weiteren EUs werden wie folgt eingeteilt: EUs, die sehr frequentiert bei praktisch jedem Problem der Aufgabenklasse verwendet werden. Als Beispiele sind hier Addition und Subtraktion zu nennen. Diese EUs werden, wie bereits alle Konvertierungs-EUs, den statischen EUs zugeteilt und damit nicht rekonfiguriert. Als letztes ist die Gruppe der aufgabenabhängig verwendeten (nicht Konvertierungs-) EUs zu nennen. Diese EUs werden nicht bei jedem Problem verwendet und benötigen zudem vergleichsweise viele Ressourcen. Die exakten Zahlen der benötigten Ressourcen können Tabelle C.1 aus Anhang C.3 entnommen werden. Die Ergebnisse dieser Diskussion werden in der folgenden Tabelle 5.2 zusammengefasst.

Um im Rahmen dieser Experimente eine sinnvolle Rekonfiguration zu ermöglichen und den Ressourcenoverhead der zu definierenden pBlocks zu minimieren, sollten diese EUs vergleichbar viele Ressourcen benötigen. Dabei ist es möglich, die benötigten Ressourcen jedes dieser EUs in einem gewissen Rahmen zu beeinflussen, indem verschiedene Eigenschaften variiert werden. Variationsmöglichkeiten bestehen in der Verwendung von BRAM- und/oder DSP-Ressourcen, sofern die jeweilige EU diese verwenden kann. Weiterhin gibt es Variationsmöglichkeiten in der Pipelinelänge, die die EU zur Berechnung des jeweiligen Ergebnisses benötigt. Damit ist also die Anzahl an Taktzyklen gemeint, die eine Berechnung dieser EU dauert. Theoretisch könnte ebenfalls das volle Pipelining der einzelnen EUs (teilweise) abgeschaltet werden, um weitere Ressourcen zu sparen. Dies ist allerdings kein für den ViSARD sinnvoller Anwendungsfall und wird daher mit Ausnahme der trigonometrischen Operationen nicht betrachtet (siehe Anhang C.1).
Bei der Veränderung der Pipelinelänge muss beachtet werden, das eine zu geringe Anzahl an Pipelinestufen zu einer starken Reduktion der maximalen Taktfrequenz führt. Diese durch den Softcore erreichbare maximale Taktfrequenz ist allerdings erst bekannt, wenn das gesamte Design feststeht und auch die Platzierung und das Routing auf dem FPGA abgeschlossen ist, also nicht im Vorfeld über die Methodologie festgelegt werden kann. Um im Rahmen dieser Experimente alle EUs auf in per Praxis sinnvoll verwendbare Pipelinelängen einzugrenzen und um eine vergleichbare Basis zu schaffen, wurde eine Frequenz von 100 MHz als untere Grenzfrequenz festgelegt.
Im Rahmen dieser Tests wurden die EUs zur Berechnung von Sinus und Cosinus auf annähernd doppelte Genauigkeit umgebaut (46 Bit Gleitkommagenauigkeit der Ausgabe). Detailtiere Informationen dazu sind in Anhang C.1 zu finden.
Dieser Schritt erfolgte nach einer Analyse (*Evaluation*) der vorhandenen Funktionalität des Softcores und entspricht dem Erstellen eines neuen Softcore-Moduls (*Create New Softcore Modul*) des Φ-Vorgehensmodells, siehe Abbildung 4.1.
Die Ergebnisse der jeweiligen Einstellmöglichkeiten aller EUs ist in Anhang C.3 zu finden. Aus den gemessenen Ergebnissen vom Ressourcenverbrauch der zu rekonfigurierbaren EUs ergeben sich folgende Schlussfolgerungen:

- Nur die natürliche Exponentialfunktion kann BRAM Ressourcen verwenden. Weiterhin benötigt diese etwa 2000 Slices (ca. 15 % des gesamten FPGAs), wenn keine DSPs

| Funktion | Operatoren | Statisch/ Rekonfigurierbar |
|---|---|---|
| Datentransformation | • Absoluter Betrag | Statisch |
| | • Single ↔ Double | Statisch |
| | • Double ↔ Integer32 | Statisch |
| | • Double ↔ Integer16 | Statisch |
| | • Double ↔ UnsignedInteger8 | Statisch |
| Kopieroperationen | • Speicherslot kopieren | Statisch |
| | • Speicherslot kopieren mit Bedingung (wenn positiv,null,...) | Statisch |
| Berechnungen | • Addition | Statisch |
| | • Subtraktion/Absolute Subtraktion | Statisch |
| | • Multiplikation | Rekonfigurierbar |
| | • Division | Rekonfigurierbar |
| | • Wurzelberechnung | Rekonfigurierbar |
| | • $e^x$-Berechnung | Rekonfigurierbar |
| | • Sinus/Cosinus-Berechnung | Rekonfigurierbar |

Tabelle 5.2: Statische und Rekonfigurierbare Operatoren

verwendet werden. Mit der DSP Verwendung werden lediglich noch ca. 1.5 % der verfügbaren Slices benötigt. Da innerhalb des ViSARD lediglich die EUs DSP Ressourcen verwenden, wird hier die Konfiguration unter voller Verwendung von DSPs genutzt. Weiterhin wird der BRAM für diese EU verwende, da diese Ressource für keine weitere Logik benötigt wird.

- Division und Sinus/Cosinus sind die (mit Abstand) größten EUs im Sinne der benötigten Slices, da diese nicht mit BRAM oder DSPs realisiert werden können.
- Die Variation der Verwendung von DSPs bei der Multiplikation hat einen geringen Einfluss auf die benötigte Anzahl an Slices sofern überhaupt DSPs verwendet werden.
- Um die schnellstmögliche Lösung in der praktischen Anwendung identifizieren zu können, sind mehrere Iterationsstufen notwendig. Eine Veränderung der Pipelinelänge des EUs mit dem kritischen Pfad kann zu einer Erhöhung der Taktfrequenz des gesamten ViSARD führen.

Die folgenden Experimente führen dabei immer die erste Iterationsstufe durch. Das Ziel der Experimente ist nicht die Ermittlung der jeweiligen Lösung mit dem schnellsten Gesamtergebnis, sondern der Nachweis der korrekten Funktionalität und der praktisch Nachweis der benötigten Rekonfigurationszeiten der einzelnen pBlocks. Diese Rekonfigurationszeiten würden sich in weiteren Iterationsstufen nicht oder nur minimal verändern, da sich die Größe des jeweiligen pBlocks und damit der notwendigen partiellen Bitströme nicht ändert.
Aus diesen Ergebnissen ergeben sich folgende, in Tabelle 5.3 dargestellte, sinnvolle Rekonfigurationspaarungen unter den zu rekonfigurierenden EUs. Dabei wird innerhalb der jeweiligen Rekonfigurationspaarungen (Paarung) zwischen den einzelnen Konfigurationen (Konfig.) unterschieden und zu jeder Konfiguration die darin enthaltenen Operatoren (Enthaltene Operatoren), sowie deren Ressourcenkonfigurationen (Ressourcenkonfiguration) dargestellt. Eine Erläuterung der Ressourcenkonfiguration kann unter Anhang C.3 gefunden werden.

| Paarung | Konfig. | Enthaltene Operatoren | Ressourcenkonfiguration |
|---|---|---|---|
| 1 | 1 | $e^x$-Operator | BRAM: ohne, DSP: Mittel |
| | 2 | Wurzel-Operator | |
| 2 | 1 | Wurzel-Operator | |
| | 2 | Sinus/Cosinus-Operator | Seriell, 48-Bit Genauigkeit |
| | 3 | Divisions-Operator | |
| 3 | 1 | $e^x$-Operator | BRAM: Ohne, DSP: Voll |
| | | Wurzel-Operator | |
| | 2 | Sinus/Cosinus-Operator | Parallel, 32-Bit Genauigkeit |
| | 3 | Divisions-Operator | |
| 4 | 1 | Divisions-Operator | |
| | 2 | Wurzel-Operator | |
| | | Sinus/Cosinus-Operator | Seriell, 48-Bit Genauigkeit |
| | 3 | $e^x$-Operator | BRAM: Voll, DSP: Mittel |
| | | Wurzel-Operator | |
| | | Multiplikations-Operator | DSP: Voll |

Tabelle 5.3: Rekonfigurationspaarungen der Operatoren

Mit allen vier Rekonfigurationspaarungen kann eine Evaluation sinnvoll ausgeführt werden. Um die korrekte Funktionsweise nachzuweisen, wurde Möglichkeit 1, der Austausch vom EU zur Berechnung der natürlichen Exponentialfunktion mit dem EU zur Berechnung der Wurzel ausgewählt. Ein weiteres Experiment mit einem Austausch mehrerer EUs wird mit Option 4 ebenfalls praktisch ausgeführt.
Da die jeweiligen EUs an die Verwendung als partiell austauschbare EUs angepasst werden müssen und die Platzierung auf dem FPGA durch den Nutzer festgelegt werden muss, werden diese als neue Softcore Module (*Create New Softcore Module*, siehe Abbildung 4.1 des Φ-Modells) betrachtet. Zur Festlegung dieser Module ist eine Auswertung (*Evaluation*) des Softcore-Modells (*Timed Softcore Model*) mit den Informationen über Timing und Ressourcenverbrauch durchgeführt worden. Dieser Schritt entspricht ebenfalls dem partiellen Rekonfigurationschecker (*pR Checker*) der praktisch realisierten Toolchain aus Abbildung 5.1.
Weiterhin wurde das Schema aus Abbildung 2.8 zur Logikplatzierung und Ressourcenabschätzung für rekonfigurierbare Teile hier verwendet. Die Operatoren wurden als mögliche rekonfigurierbare Elemente(*Reconfig. Part(s) Design*) ausgewählt und entsprechend

umgebaut. Im Anschluss erfolgte eine Ressourcenabschätzung (*Resource Estimation*) der Operatoren mit Hilfe der Vivado-Suite.
Das Testsystem für das erste Experiment enthält dabei exakt einen ViSARD Softcore Prozessor, der im Single-Core Betrieb Berechnungen durchführt, sowie einen PRCO (*PRCO*), eine Kontrollogik (*Control Logic*) und ein Bitstreamlade-Modul (*Bitstream Loader*), wie bereits in Abbildung 5.7 vorgestellt.
Das Testsetup sieht weiterhin vor, dass der Prozessor je eine Berechnung mit dem aktuell vorhandenen EU durchführt, diesen dann rekonfiguriert und eine weitere Berechnung mit dem neuen EU durchführt. Anhand der Ergebnisse der Berechnungen kann das korrekte Austauschen und Anschließen der neuen EU nachgewiesen werden. Weiterhin wird damit die korrekte Funktionsweise der in Abbildung 5.7 vorgestellten Speicherlösung der Bitstreams nachgewiesen.
Es wurde eine geeignete Platzierung der rekonfigurierbaren Teile ermittelt (*Placement Options*, siehe Abbildung 2.8). In Abbildung 5.8 ist die resultierende FPGA-Belegung dargestellt. Dabei ist der relevante (belegte) Teil des FPGAs als Ausschnitt zu sehen.

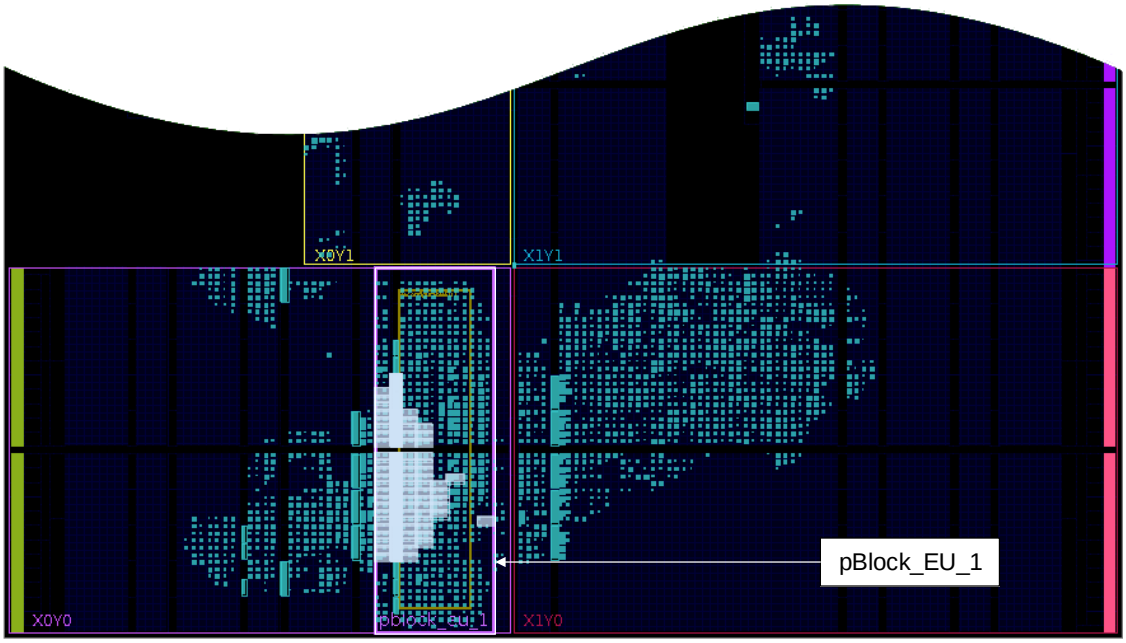



Abbildung 5.8: Resultierende Belegung des FPGAs

Zu sehen ist die definierte Region des pBlocks zum Austausch der zuvor ausgewählten EUs (in rosa) sowie die restliche Belegung des FPGAs. Alle hervorgehobenen Ressourcen des FPGAs werden dabei bei diesem Test vom Prozessor und der weiteren Logik verwendet. Es ist anzumerken, dass im Vergleich zu Abbildung 2.5, aufgrund unterschiedlicher Tool-Versionen sowohl die BRAM-Spalten (ursprünglich rot dargestellt), als auch die DSP-Spalten (ursprünglich grün dargestellt) farblich nicht mehr hervorgehoben werden und als leere schwarze Spalten dargestellt werden.
Die exakten Ressourcen der verwendeten Komponenten sind Tabelle 5.4 zu entnehmen. Um einen Vergleich anfertigen zu können sind in dieser Tabelle ebenfalls die notwendigen Ressourcen des ViSARD aufgeführt, wenn dieser in der nicht rekonfigurierbaren Version (Statischer ViSARD), also gleichzeitig mit allen notwendigen Operatoren, vorliegt.

| Komponente | LUTs | FFs | BRAM | DSPs |
|---|---|---|---|---|
| PRCO | 251 | 502 | 1 | 0 |
| Natürliche Exponentialfunktion | 2.049 | 4.098 | 0 | 15 |
| Wurzelfunktion | 1.713 | 3.426 | 0 | 0 |
| Konfiguration 1 | 2.096 | 4.192 | 0 | 15 |
| Konfiguration 2 | 1.763 | 3.526 | 0 | 0 |
| pBlock Region | 2.800 | 5.600 | 0 | 20 |
| Statischer Teil (des reconf. ViSARD) | 1.230 | 2.460 | 4 | 3 |
| Rekonfigurierbarer ViSARD (komplett) | 4.030 | 8.060 | 4 | 23 |
| Statischer ViSARD | 4.745 | 9.490 | 3 | 18 |
| Differenz statisch vs. rekonfigurierbar | -715 | -1.430 | +1 | +5 |

Tabelle 5.4: Ressourcenbedarf der Komponenten

Es ist anzumerken, dass es einen geringen Unterschied in den benötigten Ressourcen einiger Operatoren, z. B. der Wurzelfunktion, gibt. Dieser Unterschied kommt zustande, weil innerhalb der rekonfigurierbaren Region diese Ressourcen als Routing-Endpunkte für ungenutzte Partition-Pins allokiert werden müssen. Diese ungenutzten Partition-Pins existieren, weil die Operatoren der Wurzelfunktion und zur Berechnung der natürlichen Exponentialfunktion kein zweites Argument und damit keinen zweiten Input-Operanden benötigen. So ist auch der Unterschied zwischen den benötigten Ressourcen der einzelnen EUs, im Vergleich zur Konfiguration in der lediglich diese EUs verwendet werden zu erklären.
Weiterhin ist anzumerken, dass in der Tabelle 5.4 und allen weiteren Tabellen im Abschnitt der partiellen Rekonfiguration unter den Angaben der Ressourcen die Anzahl an Ressourcen zu verstehen sind, die im FPGA nicht für weitere Module zur Verfügung stehen. Das bedeutet, dass beispielsweise der PRCO zwar lediglich 294 Flip-Flops real verwendet bzw. benötigt, allerdings aufgrund der 251 verwendeten LUTs durch den PRCO insgesamt 502 FFs belegt werden. Diese Zahl ergibt sich aus der Tatsache, dass zu jeder LUT insgesamt zwei FFs fest dazugehören. Dabei ist es egal ob ein Modul wie der PRCO davon null, ein oder beide FFs je LUT verwendet, die restlichen FFs stehen anderen Modulen nicht zur Verfügung und müssen daher als Ressorucenverbrauch kalkuliert werden. Insgesamt liegt das Nutzungsverhältnis von LUT zu Flip-Flop im ViSARD bei etwa 1:0,77. Wie bereits diskutiert, werden dennoch die belegten und nicht die verwendeten Ressourcen als Berechnungsgrundlage verwendet.
Es gibt einen Spezialfall, indem es möglich ist, dass Flip-Flops durch in der direkten Umgebung befindliche LUTs verwendet werden. Voraussetzung dazu ist, dass alle beteiligten Flip-Flops identische Kontrollsignale (wie z. B. Set/Reset, Clk, etc.) erhalten und das eine entsprechende nicht näher definierte räumliche Nähe gegeben ist. In diesen Experimenten wird davon ausgegangen, dass der ViSARD eigene Kontrollsignale verwendet. [Xil16a] Weitere auf dem FPGA befindliche Module verwenden also nicht die selben Kontrollsignale wie der ViSARD und aus diesem Grund wird davon ausgegangen, dass eine Verwendung der durch den ViSARD nicht benötigter Flip-Flops nicht möglich ist.
Wie der Tabelle 5.4 entnommen werden kann, können bei einer partiellen Rekonfiguration für den Austausch der Operatoren zur Berechnung der natürlichen Exponentialfunktion mit dem Operator zur Berechnung der Wurzelfunktion 715 LUTs und 1.430 Flip-Flops eingespart werden.
Es kann also gesagt werden, dass es möglich ist bei diesem einfachen Test die für die beiden EUs

notwendigen LUTs um 26,49 % zu senken. Diese Senkung errechnet sich aus der prozentualen Differenz der LUTs des pBlocks mit der Summe der LUTs aus Wurzelfunktion und natürlicher Exponentialfunktion. Insgesamt benötigt der rekonfigurierbare ViSARD im Vergleich zum statischen ViSARD 15,07 % weniger LUTs. Das ist ein wichtiges und gutes Ergebnis, da die LUTs gerade im Einsatz als MIMD-Prozessor die einschränkende Ressource sind. Belegt wird diese Aussage mit den Ergebnissen des im Vorwort erwähnten „WLI SoC“-Projektes. In diesem Projekt kommen in dem vorgestellten Zynq 7020 insgesamt sechs ViSARD Prozessoren in einer MIMD-Konfiguration zur Abarbeitung der Algorithmen zum Einsatz. Diese verbrauchen insgesamt 82,74 % der verfügbaren LUTs. Gleichzeitig werden 17,16 % der BRAM-Ressourcen und 49,09 % der DSP-Ressourcen benötigt. Eine Reduktion der benötigten LUTs würde also dazu führen, dass weitere Softcores eingesetzt und die kombinierte Rechenleistung damit gesteigert werden könnte.
Hier ist es also möglich bei einem Austausch von diesen beiden EUs ca. 15 % mehr Prozessoren auf denselben Chip zu bekommen und so entsprechende Berechnungen durch höhere parallele Abarbeitung zu beschleunigen.

Um die Rekonfigurationszeiten messen zu können, wurde speziell für die Tests mit der partiellen Rekonfiguration eine Logikanalysator IP in das Design eingebettet.
Sämtliche Rekonfiguration konnte dabei innerhalb des Experimente erfolgreich mit folgenden Ergebnissen ausgeführt werden:

| Methode | Exponentialfunktion | | Wurzelfunktion | |
|---|---|---|---|---|
| Xilinx Standard | 227.336 Byte | 100 % | 227.336 Byte | 100 % |
| Xilinx Kompression | 195.424 Byte | 85,96 % | 190.660 Byte | 83,87 % |
| Standard ohne Blanking | 113.772 Byte | 50,05 % | 113.772 Byte | 50,05 % |
| Differenz-basiert | 113.748 Byte | 50,04 % | 113.748 Byte | 50,04 % |
| torCombitgen | 113.732 Byte | 50,03 % | 113.732 Byte | 50,03 % |

Tabelle 5.5: Bitstreamgrößen der unterschiedlichen Verfahren

Die Rekonfiguration benötigt mit dem Xilinx Standard Bitstream (siehe Tabelle 5.5) im Durchschnitt 56.880 Takte. Bei einer Frequenz von zuvor festgelegten 100 MHz entspricht das einer Rekonfigurationszeit von 568,80 $\mu$s.
Die Größe des partiellen Bitstreams betrug dabei jeweils 227.336 Bytes. Damit konnte also ein durchschnittlicher Durchsatz von 399,68 MB/s ($227.336 \text{ Bytes} \div 568,8\ \mu\text{s} = 399,68 \text{ MB}$) erzielt werden. Das entspricht einem erreichten Durchsatz von 99,92 % im Vergleich zum theoretisch maximalen Durchsatz.
Dabei ist zu beachten, dass die erreichten 56.880 Takte zur Rekonfiguration die durchschnittlich erzielten Rekonfigurationszeiten sind, mit einer durchschnittlichen Abweichung von 4,9 Takten. Im Maximum hat die gleiche Rekonfiguration 56.911 Takte benötigt. Eine Analyse der längsten Rekonfigurationszeit ergibt also analog eine maximale Rekonfigurationszeit von 569,11 $\mu$s und damit einen erreichten Durchsatz von 399,45 MB/s. Diese Abweichung kommt durch unterschiedliche Zugriffslatenzen auf den Speicher zustande. Der PRCO verhält sich absolut deterministisch und besitzt damit ein exakt vorhersehbares Timing. Es ist allerdings zu beachten, dass um die Einhaltung der harten Echtzeitschranke garantieren zu können, ein Speicher mit explizit angegebenen maximalen Latenzen verwendet wird und diese als Berechnungsgrundlage angesetzt werden.

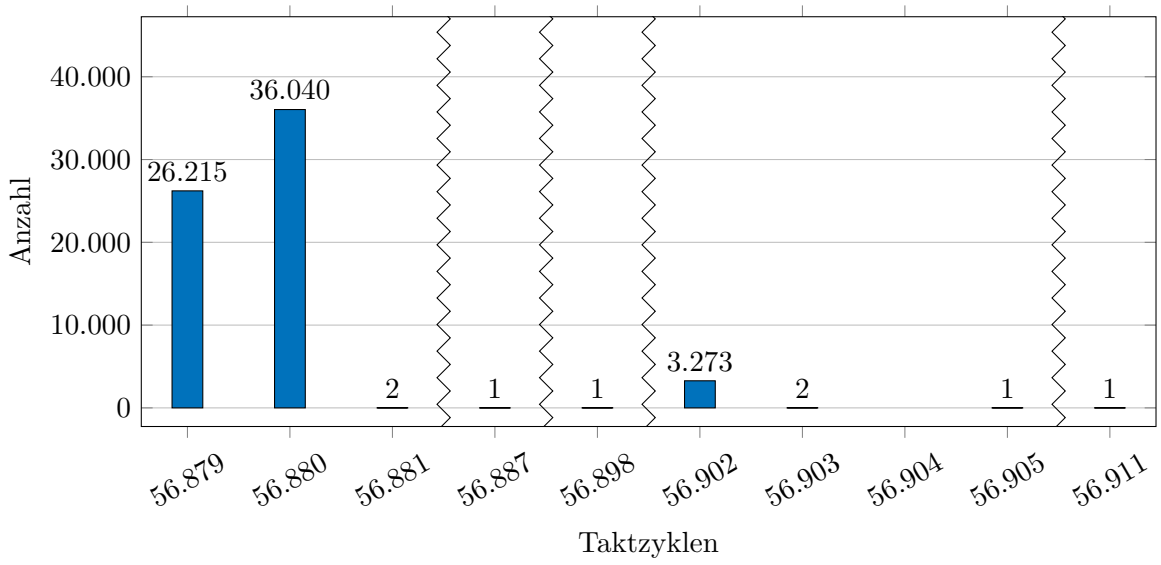



Abbildung 5.9: Verteilung der Rekonfigurationszeiten (Xilinx Standard Bitstream)

Die Rekonfigurationszeiten richten sich nach der Größe des verwendeten Bitstreams. Dabei gibt es verschiedene Möglichkeiten die Größe zu verringern. Bei dem zweiten durchgeführten Test wurde ein durch Xilinx selbst komprimierter Bitstream verwendet und die Größe des Bitstreams damit um 14,04 % verringert.

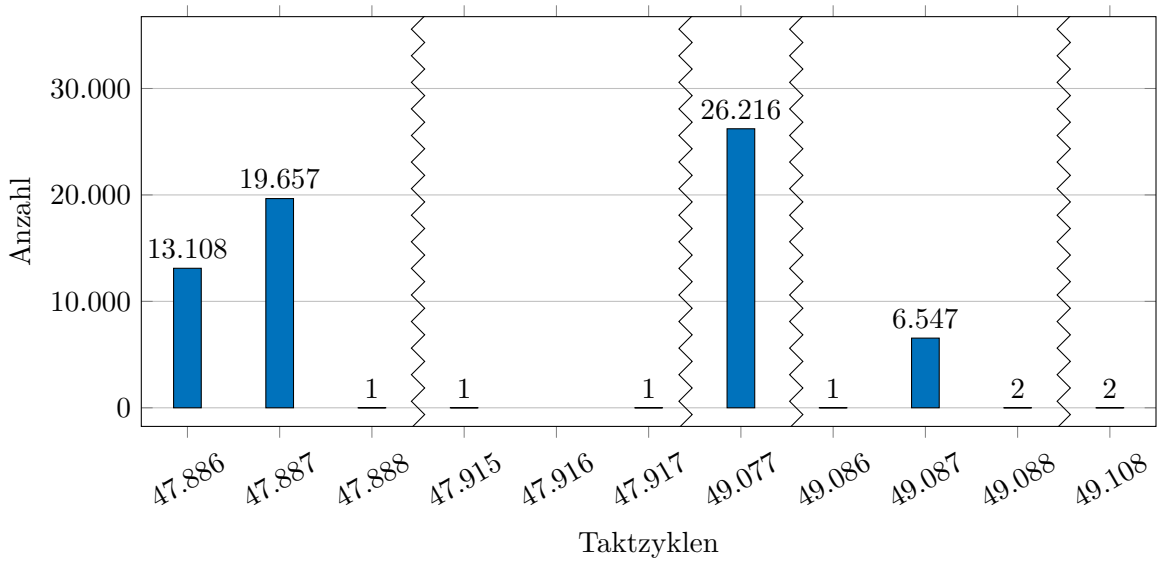



Abbildung 5.10: Verteilung der Rekonfigurationszeiten (Xilinx komprimierter Bitstream)

Wie in Grafik 5.10 zu sehen ist, kann die längste Rekonfigurationszeit von 56.911 Takten (Standard Bitstream) bereits um 7.803 Takte auf 49.108 Takte reduziert werden. Damit wird die Rekonfigurationszeit im längsten gemessenen Fall auf 491,08 $\mu$s reduziert, was einer Zeitersparnis von 13,71 % entspricht.
Die durchschnittlich erzielte Rekonfigurationszeit von 484,83 $\mu$s ergibt dabei einen

durchschnittlich erreichten Durchsatz von 398,17 MB/s. Damit konnte eine Effizienz von 99,54 % erreicht werden.

Innerhalb der Vivado-Suite ist bei der Rekonfiguration ein Problem bekannt: Die Fehler, sogenannte Glitches, können auftreten, wenn Signale der statischen Logik durch das Areal eines pBlocks laufen, während dieser Rekonfiguriert wird. Dabei spielen laut Herstellerangaben aus [Xil15] verschiedene zusätzliche, leider nicht im Detail ausgeführte, Faktoren wie inkludierte Routingressourcen, Konfigurationsframeanordnung und Übertragung des partiellen Bitstreams eine Rolle.
Bei den in dieser Arbeit verwendeten Programmen zum Erstellen und Schreiben der partiellen Bitstreams werden diese zu rekonfigurierenden Regionen mit den Bitstreams Xilinx Standard und Xilinx Kompression immer exakt zweimal geschrieben, während die anderen Verfahren lediglich das finale Schreiben realisieren und damit effektiv 50 % der Größe und damit der Rekonfigurationszeit, einsparen. Durch das zweimalige Rekonfigurieren des pBlocks kann dieser Fehler verhindert werden. Dabei wird in einem ersten Durchgang die Logik mit dem beschriebenen Blanking gelöscht und in einem weiteren Durchgang die neue Logik konfiguriert. Wie in Tabelle 5.5 zu sehen ist, kann dadurch die Größe des Bitstreams massiv, um ca. 50 %, von der in den bisher durchgeführten Experimenten verwendeten abweichen.
Es ist theoretisch möglich, die benötigte Rekonfigurationszeit damit zu halbieren. Im praktischen Einsatz des ViSARD Softcores ist dies allerdings zum jetztigen Zeitpunkt nur unter speziellen, im Folgenden erläuterten Umständen, möglich:
Sämtliche Verfahren und Bitstreams, welche die zu rekonfigurierende Logik nur einmalig beschreiben, können nur dann eingesetzt werden, wenn sichergestellt ist, dass die statische Logik davon nicht betroffen ist. Das ist dann der Fall, wenn der Softcore zum Zeitpunkt der Rekonfiguration nicht aktiv ist. Dies ist nur dann sinnvoll, wenn in einem Projekt mehrere Softcore Prozessoren parallel zu Berechnungen verwendet werden. Hier kann jeweils ein Softcore angehalten und rekonfiguriert werden. Dabei muss bei der Platzierung und dem Routing auf dem FPGA entsprechend eingestellt sein, dass keine weitere Logik durch diesen Teil des FPGAs geroutet werden darf. Um eine Aussage über die benötigten Rekonfigurationszeiten dieser Fälle zu erhalten, wurde der kleinste Bitstream, durch torCombitgen erzeugt, aus Tabelle 5.5 ebenfalls praktisch getestet. Die beiden verbleidenen Bitstreams „Standard ohne Blanking“ und „Differenz-basiert“ wurden nicht im praktischen Einsatz auf dem FPGA getestet, da die Ergebnisse redundant wären.

Wie der Grafik 5.11 entnommen werden kann, liegt die längste Rekonfigurationszeit hier bei 28.499 Takten. Damit kann die Rekonfigurationszeit, im Vergleich mit dem Xilinx Standard Bitstream, um 28.412 Takte reduziert werden. Damit ergibt sich eine resultierende Rekonfigurationszeit von 284,99 $\mu$s. Das entspricht einer Zeitersparnis im Vergleich zum originalen Bitstream von 49,93 %.
Die durchschnittlich erzielte Rekonfigurationszeit von 284,8 $\mu$s ergibt dabei einen durchschnittlich erreichten Durchsatz von 399,34 MB/s. Damit konnte eine Effizienz von 99,83 % erreicht werden.
Innerhalb des Tests mit dem torCombitgen Bitstream, bei dem die zu rekonfigurierende Region nur einmalig geschrieben wurde, traten keine Fehler wie die oben beschriebenen Glitches oder ähnliches auf. Damit kann allerdings keine Garantie gegeben werden, dass diese niemals bei der Rekonfiguration des ViSARD auftreten können. Aus diesem Grund muss, wie bereits beschrieben, der zu rekonfigurierende Softcore bei einer Rekonfiguration mit dem torCombitgen

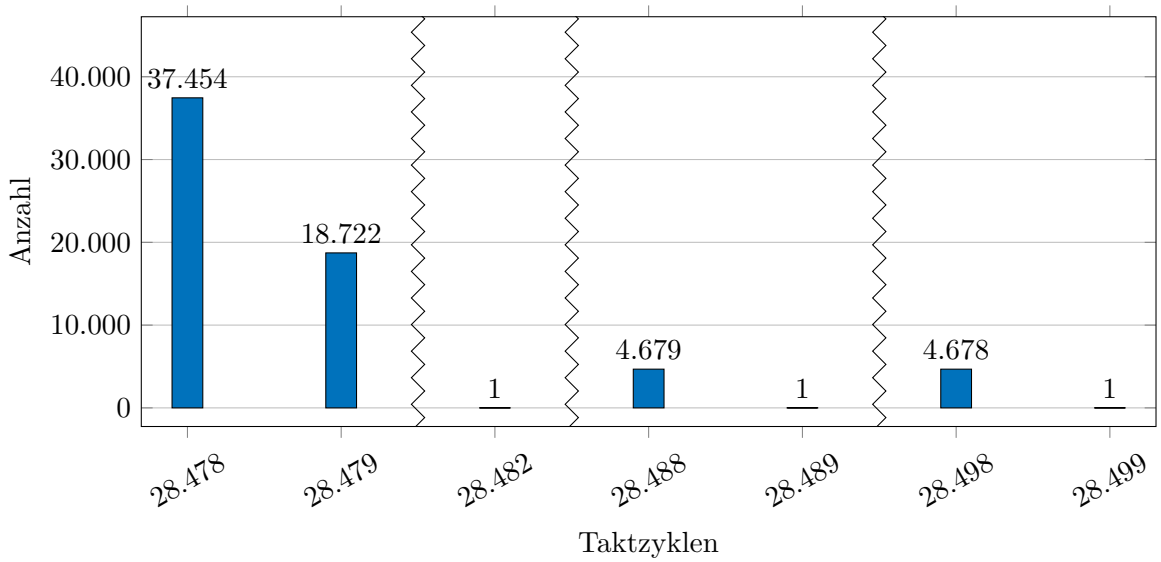



Abbildung 5.11: Verteilung der Rekonfigurationszeiten (torCombitgen Bitstream)

Bitstream angehalten werden. Die Ergebnisse der drei Abbildungen 5.9 bis 5.11 werden in der Tabelle 5.6 zusammengefasst.

| Methode | Rekonfigurationszeit (Takte) | | |
|---|---|---|---|
| | Minimum | Durchschnitt | Maximum |
| Xilinx Standard | 56.879 | 56.884 | 56.911 |
| Xilinx Kompression | 47.886 | 48.483 | 49.108 |
| torCombitgen | 28.489 | 28.480 | 28.499 |

Tabelle 5.6: Rekonfigurationszeiten

### Multi-EU Rekonfiguration in der ALU

In einem weiteren Experiment auf derselben Granularitätsebene wurden nicht zwei EUs gegeneinander getauscht, sondern das Experiment wurde erweitert auf insgesamt drei Konfigurationsmöglichkeiten und den Tausch mehrerer EUs zwischen jeder Konfiguration:

| Paarung | Konfig. | Enthaltene Operatoren | Ressourcenkonfiguration |
|---|---|---|---|
| 4 | 1 | Divisions-Operator | |
| | 2 | Wurzel-Operator | |
| | | Sinus/Cosinus-Operator | Seriell, 48-Bit Genauigkeit |
| | 3 | $e^x$-Operator | BRAM: Voll, DSP: Mittel |
| | | Wurzel-Operator | |
| | | Multiplikations-Operator | DSP: Voll |

Tabelle 5.7: Rekonfigurationspaarung Vier (Multi-EU Rekonfiguration)

Zusätzlich wurde für diesen Test der pBlock absichtlich nicht, wie zuvor vorgestellt, über die gesamte Breite der Clock-Region ausgedehnt. Ergebnis ist ein pBlock, dessen Ressourcen sehr

nah an das für die Rekonfigurationen benötigt Minimum herankommt. Getestet wird dadurch im Rahmen dieses Experiments auch zusätzlich das Auftreten zuvor beschriebener potenzieller Probleme.
Die in diesem Test benötigten Ressourcen können der folgenden Tabelle 5.8 entnommen werden.

| Komponente | LUTs | FFs | BRAM | DSPs |
|---|---|---|---|---|
| PRCO | 247 | 494 | 1 | 0 |
| Division | 3.196 | 6.392 | 0 | 0 |
| Wurzelfunktion | 1.713 | 3.426 | 0 | 0 |
| Sinus/Cossinus | 1.352 | 2.704 | 0 | 0 |
| Natürliche Exponentialfunktion | 1.121 | 2.242 | 2,5 | 15 |
| Multiplikation | 220 | 498 | 0 | 10 |
| Konfiguration 1 | 3.273 | 6.546 | 0 | 0 |
| Konfiguration 2 | 3.273 | 6.546 | 0 | 0 |
| Konfiguration 3 | 3.055 | 6.166 | 2,5 | 25 |
| pBlock | 3.800 | 7.600 | 10 | 30 |
| Statischer Teil | 1.302 | 2.604 | 4 | 3 |
| Rekonfigurierbarer ViSARD (komplett) | 5.102 | 10.204 | 14 | 33 |
| Statischer ViSARD | 8.607 | 17.272 | 5,5 | 28 |
| Differenz statisch vs. rekonfigurierbar | -3.505 | -7.068 | +8,5 | +5 |

Tabelle 5.8: Ressourcenbedarf der Komponenten (Multi-EU Rekonfiguration)

Wie in Tabelle 5.8 gesehen werden kann, werden bei drei realisierten Konfigurationen die eingesparten Ressourcen nocheinmal deutlich besser. Im ersten Test konnten insgesamt 715 LUTs gespart werden. In diesem Test konnten bereits 3.505 LUTs eingespart werden. Damit konnten, im Vergleich zum statischen ViSARD 40,72 % der benötigten LUTs eingespart werden.
Ein weiteres Ergebnis dieses Experiments ist, dass die Rekonfiguration trotz des minimierten pBlocks funktioniert hat.
Die resultierenden Bitstreamgrößen können der folgenden Tabelle 5.9 entnommen werden.

| Methode | Konfig. 1 | | Konfig. 2 | | Konfig. 3 | |
|---|---|---|---|---|---|---|
| Xilinx Standard | 494.104 Byte | 100 % | 494.104 Byte | 100 % | 494.104 Byte | 100 % |
| Xilinx Kompression | 347.276 Byte | 70,28 % | 347.276 Byte | 70,28 % | 447.960 Byte | 90,66 % |
| Standard ohne Blanking | 247.156 Byte | 50,02 % | 247.156 Byte | 50,02 % | 247.156 Byte | 50,02 % |
| Differenz-basiert | 247.132 Byte | 50,02 % | 181.748 Byte | 36,78 % | 247.132 Byte | 50,02 % |
| torCombitgen | 247.116 Byte | 50,01 % | 247.116 Byte | 50,01 % | 247.116 Byte | 50,01 % |

Tabelle 5.9: Bitstreamgrößen der unterschiedlichen Verfahren (Multi-EU Rekonfiguration)

Die dabei benötigten Rekonfigurationszeiten werden in Tabelle 5.10 dargestellt. Die exakten grafischen Abbildungen zu den Rekonfigurationszeiten sind in Anhang C.4 unter den Abbildungen

C.3 bis C.5 zu finden. Hierbei wurden ebenfalls drei Bitstreams praktisch getestet: der Xilinx Standard Bitstream, der durch Xilinx komprimierte Bitstream und der durch das eigene Tool torCombitgen erzeugte Bitstream.

| Methode | Rekonfigurationszeit (Takte) | | |
|---|---|---|---|
| | Minimum | Durchschnitt | Maximum |
| Xilinx Standard | 123.571 | 123.573 | 123.598 |
| Xilinx Kompression | 86.864 | 95.256 | 112.046 |
| torCombitgen | 61.824 | 61.834 | 61.854 |

Tabelle 5.10: Rekonfigurationszeiten (Multi-EU Rekonfiguration)

Aus diesen Ergebnissen lassen sich erneut die maximal benötigten Rekonfigurationszeiten berechnen:

- Xilinx Standard Bitstream: 1.235,98 $\mu$s
- Xilinx Kompression Bitstream: 1.120,46 $\mu$s
- torCombitgen Bitstream: 618,54 $\mu$s

Die durchschnittlich erzielte Rekonfigurationszeiten von 1.235,73 $\mu$s (Xilinx Standard Bitstream), 952,56 $\mu$s (Xilinx Kompression Bitstream) und 618,34 $\mu$s (torCombitgen Bitstream) ergeben dabei durchschnittlich erreichte Durchsätze von 399,84 MB/s; 399,80 MB/s bzw. 399,64 MB/s. Damit konnte eine Effizienz von 99,96 % (Xilinx Standard Bitstream), 99,95 % (Xilinx Kompression Bitstream) bzw. 99,91 % (torCombitgen Bitstream) erreicht werden.

#### Rekonfiguration eines gesamten Softcore Prozessors

Um die Ergebnisse der durchgeführten Experimente auf der feinsten Rekonfigurationsgranularität verifizieren zu können, wurden weitere Experimente durchgeführt. Bei diesen wurde eine gröbere Rekonfigurationsgranularität gewählt und entsprechend wurde der komplette Softcore Prozessor rekonfiguriert. In diesem Rahmen wurden zwei Konfigurationen erstellt. Der Unterschied in den Konfigurationen liegt, wie bereits im ersten Test, darin, dass Konfiguration Eins den Wurzeloperator und Konfiguration Zwei den Operator der natürlichen Exponentialfunktion beinhaltet. Zusätzlich wurden die gesamten Daten der Programm- und Datenspeicher der jeweiligen Konfigurationen ausgetauscht. Dieses Szenario stellt den Wechsel des zu berechnenden Algorithmus des gesamten Softcore Prozessors dar.
Die Rekonfiguration des gesamten Softcore Prozessors entspricht dem möglichen in [Cla11] vorgestellten Einsatzszenario: Der Prozessor kann als MIMD-Multi-Prozessorsystem zur Verarbeitung von Bildern im Rahmen eines Fahrassistenzsystems eingesetzt werden. Eine Rekonfiguration wird dabei notwendig, wenn ein Wechsel der Szenerie, beispielsweise durch das Einfachren des Autos in einen Tunnel, notwendig wird.
Der folgenden Tabelle 5.11 können die benötigten Ressourcen für dieses Experiment entnommen werden.

| Komponente | LUTs | FFs | BRAM | DSPs |
|---|---|---|---|---|
| PRCO | 277 | 554 | 1 | 0 |
| Konfiguration 1 | 2.622 | 5.244 | 2 | 3 |
| Konfiguration 2 | 2.955 | 5.910 | 2 | 18 |
| Statischer Teil | 310 | 620 | 1 | 0 |
| pBlock Region | 3.200 | 6.400 | 10 | 20 |
| Rekonfigurierbarer ViSARD (komplett) | 3.510 | 7.020 | 11 | 20 |
| Statischer ViSARD | 4.745 | 9.490 | 3 | 18 |
| Differenz statisch vs. rekonfigurierbar | -1.235 | -2.470 | +8 | +2 |

Tabelle 5.11: Ressourcenbedarf der Komponenten bei kompletter Softcore Rekonfiguration

Die resultierenden Bitstreamgrößen werden in Tabelle 5.12 veranschaulicht.

| Methode | Konfiguration 1 | | Konfiguration 2 | |
|---|---|---|---|---|
| Xilinx Standard | 383.344 Byte | 100 % | 383.344 Byte | 100 % |
| Xilinx Kompression | 233.348 Byte | 60,87 % | 233.348 Byte | 60,87 % |
| Standard ohne Blanking | 191.784 Byte | 50,03 % | 191.784 Byte | 50,03 % |
| Differenz-basiert | 140.940 Byte | 36,77 % | 140.940 Byte | 36,77 % |
| torCombitgen | 140.424 Byte | 36,63 % | 140.424 Byte | 36,63 % |

Tabelle 5.12: Bitstreamgrößen der unterschiedlichen Verfahren bei kompletter Softcore Rekonfiguration

Wie der Tabelle 5.11 entnommen werden kann, werden 1.235 LUTs eingespart. Zum Vergleich: im ersten Test, bei dem lediglich die zwei EUs ausgetauscht wurden, konnten lediglich 715 EUs eingespart werden. Und das, obwohl zusätzlich zu den EUs bei diesem Test der Programm- und Datenspeicher im Rahmen des Austauschs des gesamten Softcores vorgenommen wurde. Damit konnte eine Einsparung von 26,03 % erreicht werden. Das ist eine Steigerung von weiteren 10,96 % im Vergleich zum Tausch derselben EUs als einzelne Rekonfigurationen. Der Anstieg der Effizienz liegt darin begründet, dass bei der größeren zu rekonfigurierenden Region der dafür festgelegte pBlock einen wesentlich geringeren Overhead an zu reservierenden Ressourcen hat, verglichen mit dem pBlock aus dem ersten Experiment (Vergleich mit Tabelle 5.4). Je weniger Ressourcen die zu rekonfigurierenden Teile benötigen, desto größer wird der potenzielle Overhead des festzulegenden pBlocks. Allerdings sinkt mit sinkenden Ressourcen auch die Rekonfigurationszeit, da die notwendigen Bitstreams kleiner werden.

In der folgenden Tabelle 5.13 werden die gemessenen Rekonfigurationszeiten dargestellt. Die exakten grafischen Abbildungen zu den Rekonfigurationszeiten sind ebenfalls in Anhang C.4 unter den Abbildungen C.6 bis C.8 zu finden. Hierbei wurden, wie in den Experimenten zuvor, drei Bitstreams praktisch getestet: der Xilinx Standard Bitstream, der durch Xilinx komprimierte Bitstream und der durch das eigene Tool torCombitgen erzeugte Bitstream.

| Methode | Rekonfigurationszeit (Takte) | | |
|---|---|---|---|
| | Minimum | Durchschnitt | Maximum |
| Xilinx Standard | 95.881 | 95.885 | 95.916 |
| Xilinx Kompression | 58.382 | 58.383 | 58.413 |
| torCombitgen | 35.151 | 35.158 | 35.165 |

Tabelle 5.13: Rekonfigurationszeiten bei kompletter Softcore Rekonfiguration

Die benötigte Rekonfigurationszeiten von 351,65 $\mu$s bis 959,16 $\mu$s (im Vergleich mit 284,99 $\mu$s bis 569,11 $\mu$s) sind damit um 23,39 % bis 68,54 % länger als die benötigten Rekonfigurationszeiten des ersten Experiments, bei dem lediglich die einzelnen EUs rekonfiguriert wurden. Dieser Unterschied ergibt sich aus der größeren zu rekonfigurierenden Fläche und den daraus resultierenden größeren Bitstreams.
Die durchschnittlich erzielte Rekonfigurationszeiten von 958,85 $\mu$s (Xilinx Standard Bitstream), 583,83 $\mu$s (Xilinx Kompression Bitstream) und 351,58 $\mu$s (torCombitgen Bitstream) ergeben dabei durchschnittlich erreichte Durchsätze von 399,79 MB/s; 399,68 MB/s bzw. 399,40 MB/s. Damit konnte eine Effizienz von 99,95 % (Xilinx Standard Bitstream), 99,92 % (Xilinx Kompression Bitstream) bzw. 99,85 % (torCombitgen Bitstream) erreicht werden.

### 5.1.5 Fazit

Der ViSARD Softcore stellt aufgrund seiner Architektur einen Softcore Prozessor dar, der sehr effizient in der adressierten Aufgabendomäne der echtzeitkritischen Bild- und Datenverarbeitung eingesetzt werden kann. Der Prozessor und die im Rahmen des gesamten Abschnitt 5.1 vorgenommen Anpassungen demonstrieren die Funktionsweise des vorgestellten Φ-Vorgehensmodells, sowie der Spezialisierung einer Softcore-Bibliothek (*Softcore Library*) zu einem angepassten Softcore (*Customized Softcore*) unter Verwendung der partiellen Rekonfiguration (*pR Checker*) der realisierten Toolchain aus Abbildung 5.1.
Aufgrund seiner hohen Genauigkeit kann der Softcore Algorithmen mit großer Präzision ausführen. Dabei ist die Echtzeitfähigkeit dank der zugrunde liegenden Architektur des Prozessors und dem Aufbau des Assemblercodes in jedem Fall sichergestellt und muss nicht aufwendig bei jedem Projekt erneut nachgewiesen werden. Der ViSARD Softcore liegt als generisch anpassbarer applikationsspezifischer Prozessor mit verschiedenen Konfigurationsmöglichkeiten (*Softcore Library*) vor. Er kann in zwei verschiedenen Genauigkeitsstufen und sowohl als Single-Core, aber auch als Multi-Core, mit theoretisch beliebig vielen Kernen betrieben werden.
Eine wichtige Eigenschaft ist die partielle Rekonfigurierbarkeit des Softcores. Es können einzelne EUs der ALU, ganze Kerne oder komplette Softcore Prozessoren rekonfiguriert werden, um diesen an sich ändernde Algorithmen im laufenden Betrieb optimal anzupassen. Hierzu wurden Experimente mit verschiedenen Rekonfigurationsgranularitäten durchgeführt. Die Ergebnisse zeigen, das eine Rekonfiguration verschiedener Teile des Softcores bis hin zur Rekonfiguration des gesamten Softcores möglich sind. Die dabei benötigten Rekonfigurationszeiten sind dicht an den theoretisch maximal möglichen Rekonfigurationszeiten mit durchschnittlich 99,87 % des theoretischen Maximums. Dies wird aufgrund der effizienten Architektur der Teile erreicht, welche die Rekonfiguration steuern und entsprechende Informationen aus dem Speicher laden. Damit ist gezeigt, dass die in der Methodologie aus Abschnitt 4.4.4 in Gleichung 4.1 und der zugehörigen Erläuterung vorgestellte Abschätzung der notwendigen Bitstreamgröße und der daraus berechneten Rekonfigurationszeit verwendet werden kann, um Abzuschätzen ob bei den jeweiligen Projektvorgaben eine partielle Rekonfiguration die Echtzeitschranke einhalten kann. Diese Abschätzung kann ohne notwendige praktische Realisierung der partiellen Rekonfiguration der entsprechenden Teile erfolgen, was entsprechend Zeiteffizient ist.
Auf die in Abschnitt 2.6 vorgestellten Probleme der echtzeitkritischen Daten- und Bildverarbeitung, im Speziellen die AutoVision SoC Architektur, angewandt, ergibt sich Folgendes:
Bei dem dort definierten Problem müssen im Rahmen eines Fahrassistenzsystems insgesamt 31 Bilder pro Sekunde verarbeitet werden. Das ergibt eine harte Echtzeitschranke bei der Verarbeitungszeit je Bild von 32,25 ms. Ein Wechsel der Szenerie, beispielsweise durch das Einfahren des Autos in einen Tunnel, machen einen kompletten Wechsel des zugrunde liegenden Algorithmus samt Berechnungseinheiten notwendig. Die für den ViSARD gemessene Rekonfigurationszeit beim Austausch des gesamten Softcores ergab eine benötigte Zeit von 351,65 $\mu$s. Das bedeutet, dass die Rekonfiguration des gesamten Softcore Prozessors lediglich 1,09 % der Zeitspanne benötigt, die zur Rekonfiguration und anschließenden Verarbeitung des Bildes zur verfügung stehen. Damit bleiben weitere 98,91 % der Zeitspanne, also 31,89 ms, zur eigentlichen Verarbeitung des Bildes.
Durch die Rekonfiguration konnte im Rahmen dieses Experiments über 26 % der kritischen Ressource eingespart werden. Die Ergebnisse zeigen, dass eine partielle Rekonfiguration von Softcore Prozessorsystemen in der Praxis durchaus relevant und verwendbar ist. Die zeitlichen Kosten der Rekonfiguration führen, in den entsprechenden Anwendungsfällen, zu keiner Verlet-

zung der Echtzeitkriterien, sorgen aber für eine große Reduktion der benötigten Logikressourcen und ermöglichen dadurch die Verwendung kleinerer, kostengünstigerer FPGAs.

## 5.2 Modellbasierte Codegenerierung aus Datenflussgraphen

Der Modelbased Assembly Code Generator (*MACG*) ist ein Matlab/Simulink basiertes Plugin und ermöglicht es dem Nutzer für den ViSARD Softcore speziell angepassten Assemblercode zu erzeugen, ohne die speziellen Mechanismen und Eigenschaften des Assemblercodes kennen zu müssen. Der MACG wurde bereits unter [KWS+18] veröffentlicht. Der Nutzer kann mit Hilfe dieses Plugins einen ihm vorgegebenen Algorithmus als Datenflussgraph modellieren. Alle weiteren Schritte bis zur Codeerzeugung werden automatisiert von dem Tool übernommen.
Das Tool selbst arbeitet nach dem Prinzip der modellgetriebenen Softwareentwicklung (*MDSD*) und bietet entsprechend Vorteile dieser. [SEHV12]
Innerhalb des Plugins wurde bei dem Design auf eine klare Trennung zwischen Modell und interner Darstellung geachtet. Dies ermöglicht eine effiziente Bearbeitung der Modellstruktur und erlaubt das einfache Hinzufügen von weiteren Elementen. Dieser Aspekt der Erweiterbarkeit und Anpassbarkeit des Plugins an künftigte Versionen des Softcores sichern eine Kompatibilität ohne der Notwendigkeit implementierungstechnische Details im Tool selbst vornehmen zu müssen. Weiterhin ist dadurch die Nutzung virtueller Blöcke und deren Umformung in atomare Strukturen intern möglich. Unter virtuellen Blöcken werden in diesem Zusammenhang Blöcke verstanden, die keine eins zu eins repräsentation im Assemblercode besitzen, sondern intern aus mehreren atomaren Blöcken aufgebaut sind. Als Beispiele sind hier die in Abschnitt 4.5.1 unter Abbildung 4.16 vorgestellten Subsysteme (*Sub Systems*), oder auch Schleifenblöcke, wie in Abbildung 4.17 zu sehen, zu nennen. Diese Blöcke ermöglichen das effiziente Erstellen von Modellen, da hier bereits vorimplementierte Logik als Black-Box Module hinzugefügt und direkt verwendet werden kann und werden in der folgenden Abbildung 5.12 als *Predefined Models* dargestellt.

Die Abbildung 5.12 stellt eine Verfeinerung des in Abbildung 5.1 dargestellten Blocks des MACG dar, mit auszugsweise Darstellung angrenzender Blöcke. In der Abbildung 5.12 ist die schrittweise Verarbeitung dargestellt, beginnend mit der projektspezifischen Aufgabe (*Task*) bis zur Erzeugung des Assemblercodes (*Assembly Code*). Dabei wird in einem ersten Schritt ein Datenflussmodell auf Matlab/Simulink-Basis (*Matlab/Simulink Model*) unter Einfluss der definierten Rahmenbedingungen (*Constraints* und *Requirements*) und der projektspezifischen Aufgabe (*Task*) erstellt. Dieses kann mit Hilfe von bereits zuvor erstellten Black-Box Modulen (*Predefined Models*) erstellt werden, was eine Wiederverwendung von bereits erzeugten Datenflussmodellen und Teilmodellen ermöglicht. Es ist weiterhin möglich, (Teil-)Modelle des in diesem Rahmen erzeugten Datenflussmodells als Black-Box Module abzuspeichern, um diese in diesem oder zukünftigen Modellen einzusetzen.
Nach der Erstellung des Modells wird dieses über ein dafür erstelltes Frontend (*MACG Frontend*) in einen Multi-Graphen (*Multi-Graph*), wie unter Abschnitt 4.5.1 definiert, übersetzt. Dieser Multi-Graph kann nun entweder direkt in Assemblercode übersetzt werden oder mittels des Optimierers (*MACG Graph-Optimizer*) umgeformt und vereinfacht werden. Dabei kommen die in Abschnitt 4.5.2 vorgestellten Optimierungsverfahren zum Einsatz. Die Ermittlung der Pfade durch den Multi-Graphen sowie die Optimierung des Graphen sind dabei modular von anderem Programmcode getrennt, was eine einfache Anpassung und Erweiterbarkeit zur Folge hat.
Der optimierte Graph (*Optimized Multi-Graph*) wird nun durch den Assemblercodegenerator

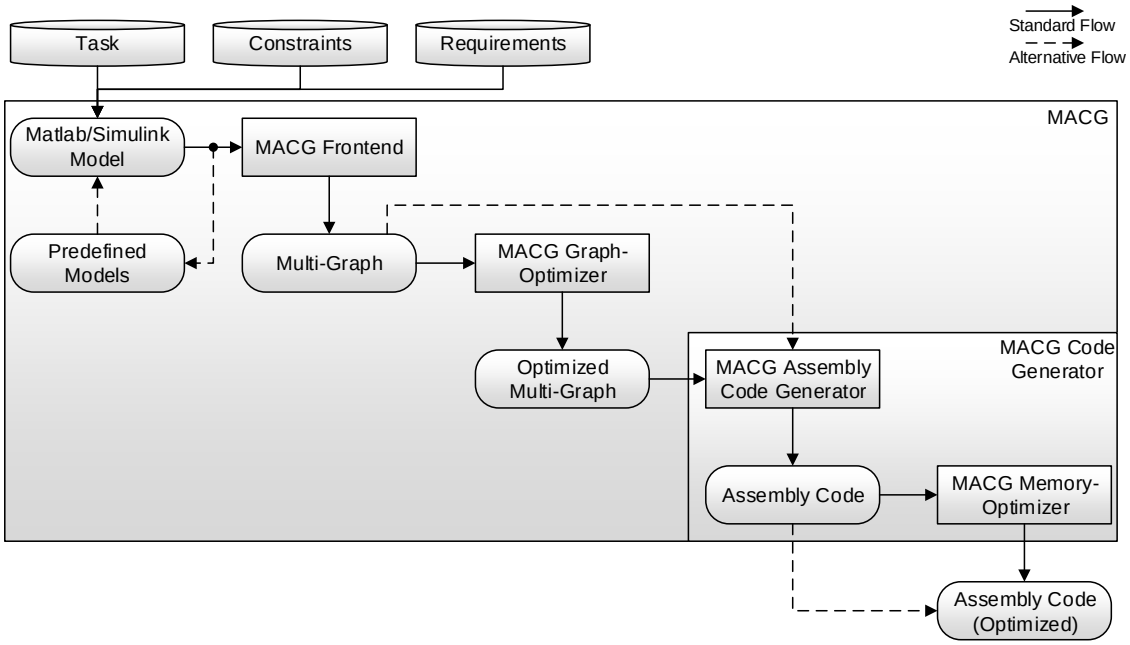



Abbildung 5.12: Workflow des MACG

(*MACG Assembly Code Generator*), unter Zuhilfenahme der in Abschnitt 4.5.3 definierten Optimierungs- und Sequenzialisierungsverfahren, in Assemblercode (*Assembly Code*) übersetzt. Dieser Assemblercode (*Assembly Code*) wird in einem weiteren Schritt mittels des Speicheroptimierers (*MACG Memory-Optimizer*) weiter optimiert. Der Speicheroptimierer setzt dabei das in Abschnitt 4.5.4 vorgestellte Optimierungsverfahren um und minimiert den benötigten Speicherverbrauch des Programms. Theoretisch besteht die Möglichkeit, diesen Schritt der Optimierung nicht durchzuführen und den zuvor erzeugten Assemblercode direkt zu verwenden. Das wird aber nicht empfohlen, da der nicht speicheroptimierte Assemblercode eine für jede deklarierte Operation eine zusätzliche Variable benötigt. Das Ergebnis des MACG ist also ein Assemblercode (*Assembly Code*), der intern Variablen verwendet. Es existiert zu diesem Zeitpunkt noch kein Mapping von Variablen auf physischen Speicher.
Bevor ein Datenflussmodell in einen Multi-Graphen umgewandelt werden kann, muss die Struktur des Modells verflacht werden. Hierzu müssen alle Systeme, sowie deren Inport- und Outport-Blöcke durch deren äquivalente Logik ersetzt werden. Die Inport- und Outport-Blöcke dienen dabei intern als Verbinder der jeweiligen Subsysteme mit der umgebenden Logik. Auf Basis dieses verflachten Modells kann nun ein Pfad gesucht werden, bei dem alle notwendigen Blöcke einbezogen werden. Ein Block kommt nur dann als Nachfolger in Betracht, wenn alle Vorgänger bereits besucht wurden. Dabei ist es eine Grundvoraussetzung, dass das Modell keine Zyklen enthalten darf.

### 5.2.1 Funktionsumfang und Erweiterbarkeit

Der Funktionsumfang des MACG besteht derzeit aus 29 verschiedenen Blöcken, die sich auf fünf verschiedene Funktionsgruppen aufteilen, wie in Tabelle 5.14 dargestellt wird.

| Funktionsgruppe | Blöcke |
|---|---|
| Eingabe/Ausgabeblöcke | • Input<br>• Output |
| Subsystemblöcke | • Subsystemblock<br>• Schleifenblock (Rollout-Block) |
| Datentransformationsblöcke | • Absoluter Betrag<br>• Single ↔ Double<br>• Double ↔ Integer32<br>• Double ↔ Integer16<br>• Double ↔ UnsignedInteger8 |
| Datenschiebeblöcke | • Speicherslot kopieren<br>• Speicherslot kopieren mit Bedingung (wenn positiv,null,...) |
| Operationsblöcke | • Addition<br>• Subtraktion/Absolute Subtraktion<br>• Multiplikation<br>• Division<br>• Wurzelberechnung<br>• Berechnung der natürlichen Exponentialfunktion |

Tabelle 5.14: Funktionsumfang des MACG

Die Bibliothek ist unter Anhang D Abbildung D.1 zu sehen.
Um die Aktualität des Plugins für künftige Erweiterungen und Veränderungen des ViSARD Softcore Prozessors zu gewährleisten, ist es vorgesehen, dass die Funktionsbibliothek erweiterbar, bzw. veränderbar ist. Um einen neuen Funktionsblock hinzuzufügen, müssen in einem ersten Schritt folgende Eigenschaften definiert werden:

- Blocktyp

- Priorität
- Assembleroperation

Für den Blocktyp muss eine Abbildung auf einen Assemblerbefehl existieren, da der von Simulink verwendete „BlockType“ nur einen lesenden Zugriff hat. Dieses Attribut kann mittels des Maskeneditors hinzugefügt werden. Die Priorität des Blocks ist für verschiedene Sequenzialisierungsstrategien notwendig, um je nach Strategie eine verbesserte Assemblercodeerzeugung zu ermöglichen. Als drittes muss der Assemblerbefehl angegeben werden, der durch den Block repräsentiert wird. Innerhalb dieses Arguments werden auch die Anzahl an Eingabe und Ausgabeparametern des Befehls festgelegt, so kann ein Befehl in der dem ViSARD zugrunde liegenden Assemblersprache null bis zwei Eingabeparameter und null oder einen Ausgabeparameter verwenden.

### 5.2.2 Sequenzialisierungsstrategien und Optimierungen

Ist das Datenflussmodell erstellt und der darauf aufbauende Graph nach den in Abschnitt 4.5.2 vorgestellten Optimierungsverfahren optimiert worden, ist der nächste Schritt die Sequentialisierung und damit Erstellung einer Reihenfolge des resultierenden Assemblercodes. Exemplarisch wird in Abbildung 5.13 ein einfaches Datenflussmodell gezeigt, was mit dem MACG erstellt wurde:

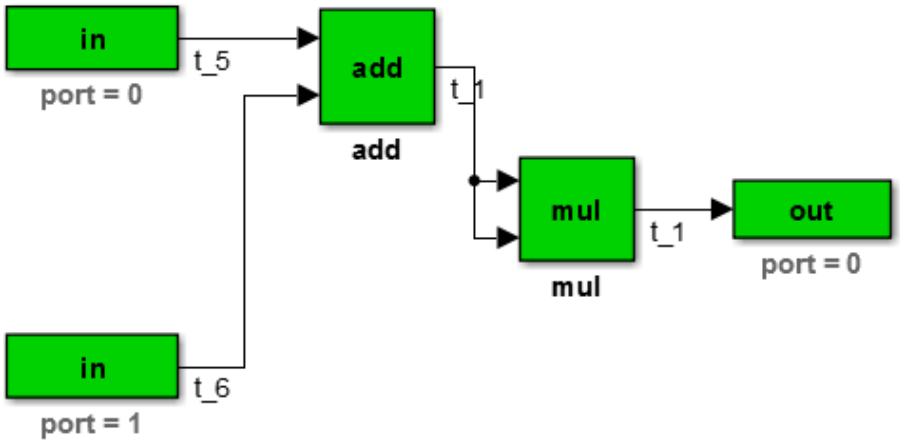



Abbildung 5.13: MACG Datenflussmodell Beispiel

Der MACG bietet verschiedene Sequenzialisierungsstrategien. Die wichtigste Strategie, die „Rollout Priority“ soll im Folgenden im Detail vorgestellt werden. Die Rollout Priority Strategie geht von der Annahme aus, dass der Nutzer logisch zusammenhängende Logikteile, also Blöcke und Teilmodelle, auch innerhalb des Modells räumlich nah aneinander anordnet. Dieser Annahme folgend, wird bei dieser Strategie über die Lage der Blöcke im ursprünglichen Modell versucht, die Anordnung im resultierenden Code so auszurollen, dass diese Logikteile möglichst nah beieinander platziert werden. Das wird an der folgenden Abbildung 5.14 verdeutlicht.

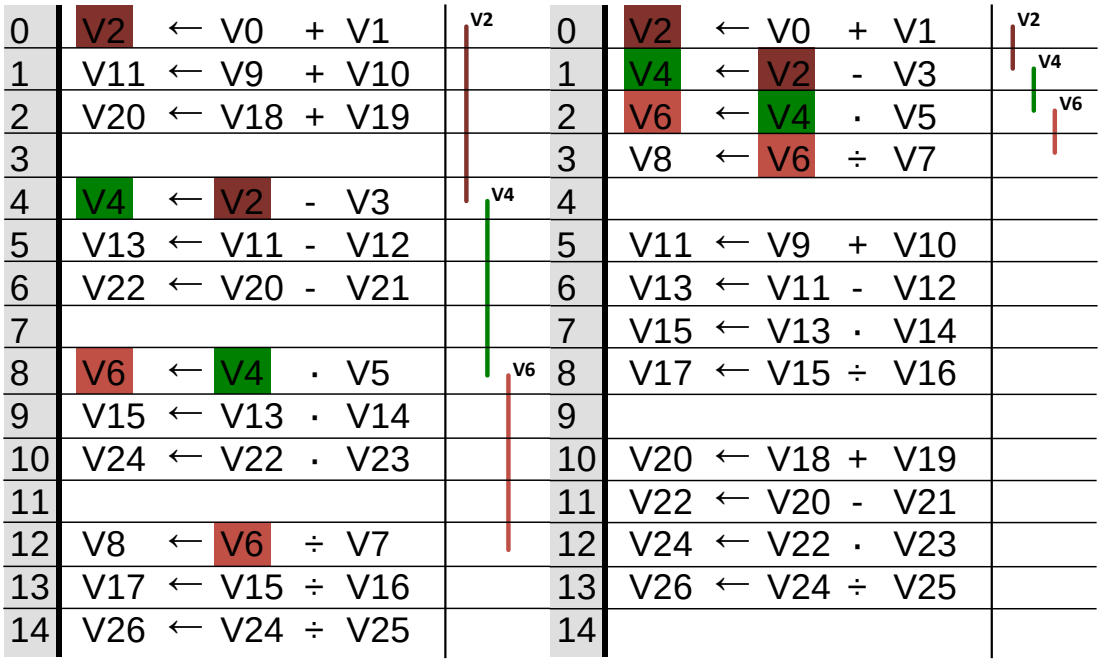



Abbildung 5.14: Variablen-Lebenszeitenvergleich Prioritätsbasiert vs. Rollout Priority

Zu sehen ist auf der rechten Seite eine Sequenzialisierung nach der Rollout Priority-Strategie. Zum Vergleich ist auf der linken Seite eine Sequenzialisierung mit einer Sortierung nach Operationstypen zu sehen. Diese Sortierung kann erreicht werden, wenn das Attribut der Priorität jedem Operator einen eineindeutigen Wert zuweist und damit eine prioritätsbasierte Strategie verwendet wird.
In Abbildung 5.14 ist zu sehen, dass die Lebenszeiten der jeweiligen Variablen mit der Rollout Priority-Strategie kürzer sind, was eine Verbesserung der Speicheroptimierungsmöglichkeiten des darauffolgenden Matrixoptimierers zur Folge hat.
Diese Strategie geht davon aus, dass in den meisten Modellen mehrere Iterationen von Befehlsabfolgen existieren, wobei die untereinander verwendeten Variablen wiederverwendet werden können, was die Speicherauslastung verringert. Das ist immer dann der Fall, wenn durch den MACG ein Schleifenkonstrukt ausgerollt wird.
Weitere Sequenzialisierungsstrategien, die hauptsächlich als Vergleichsgrundlage dienen, sind die Strategien „FiFo“, die mittels einer Liste an wählbaren Operationen immer die Operation wählt, die sich an erster Stelle der Liste befindet, oder „Random Choice“, die aus der Liste eine Operation zufällig auswählt.
Es ist ebenfalls möglich, die Auswahl manuell zu treffen, mit der Strategie „Interactive“, was bei großen Datenflussmodellen nicht empfohlen wird. Eine weitere Strategie ist die prioritätsbasierte Sequenzialisierungsstrategie, welche die entsprechende Liste nach dem in den Blöcken definierten Attribut der Priorität sortiert und daraus eine Sequenzialisierung erstellt.
Der MACG hat zwei Optimierungsziele:

- Primäre Optimierungsziel: Minimierung der benötigten Variablen (und damit des minimal benötigten Speichers) im resultierenden Assemblercode
- Sekundäres Optimierungsziel: Minimierung der Laufzeit des resultierenden Assemblercodes

Die Minimierung der Laufzeit wird dabei lediglich als sekundäres Optimierungsziel angenommen, da davon ausgegangen wird, dass die Echtzeitschranke bekannt ist. Ein Einhalten der Echtzeitschranke kann damit nachgewiesen werden und ein weiteres verkürzen der Laufzeit hat keine Notwendigkeit. Das Minimieren der Variablen führt allerdings zu einer Minimierung des notwendigen Speichers, was mit einer Kostenreduktion einhergeht. Hier hat jede weitere Reduktion einen positiven Effekt auf das Gesamtergebnis.

#### Aufbau des generierten Assemblercodes

Der durch den *MACG Assembly Code Generator* erzeugte Assemblercode ist in drei Teile eingeteilt:

- Kopfzeilen mit Modell- und Optimierungsinformationen
- Variablen- und Konstantendeklarationsabschnitt
- Algorithmusabschnitt

Innerhalb der Kopfzeile werden Informationen über den Ersteller gespeichert sowie das aktuelle Datum. Bei aktivierten Optimierungen werden diese ebenfalls mit ihren Parametern aufgelistet. Dies soll die Reproduzierbarkeit sicherstellen und die Nachvollziehbarkeit erhöhen.
Innerhalb des Variablen- und Konstantendeklarationsabschnitts werden alle verwendeten Konstanten und Variablen deklariert. Hierzu muss ebenfalls ermittelt werden, mit welchen Werten diese initialisiert werden müssen. Sollte eine Variable ihre erste Belegung durch einen Input-Block erhalten, so wird diese mit einem festgelegten Startwert initialisiert.
Der Algorithmusabschnitt legt die eigentliche Funktionalität des Assemblercodes fest. Hier werden, je nach Sequenzialisierungsstrategie, die einzelnen Befehle der Blöcke sequentiell aufgelistet. Ein exemplarischer Assemblercode mit der hier vorgestellten Einteilung ist in Anhang D.1 unter Abbildung D.2 zu finden.

### 5.2.3 Experimentelle Tests und Ergebnisse

Um die Möglichkeit zu haben, den vom MACG erzeugten Assemblercode experimentell testen zu können, wurde ein Simulator geschrieben, welcher den gegebenen Assemblercode mit etwaigen externen Eingaben simulieren kann. Dieser Schritt ist sinnvoll, da ohne eine entsprechende Simulationsmöglichkeit jeder Assemblercode zuerst in Maschinencode übersetzt und anschließend im Softcore selbst bzw. in den internen Simulationsumgebungen von Xilinx mit dem Softcore getestet werden müsste. Sollte der Softcore dabei entsprechend angepasst werden, würden weitere potenzielle Fehlerquellen hinzukommen, was eine Fehlersuche erschweren würde. Weiterhin dauert eine entsprechende Simulation des gesamten Prozessors wesentlich länger, als die Simulation der einzelnen Befehle mittels des Simulators. Aus diesem Grund ist es sinnvoll einen korrekt arbeitenden Simulator zu verwenden, um die möglichen Fehlerquellen auf Designfehler im Modellierungsprozess bzw. Fehler im MACG (sofern an diesem gearbeitet wird) beschränken zu können.
Dieser Simulator ist in Python als Kommandofenster-Tool realisiert und simuliert einen gegebenen Assemblercode, indem die dortigen Befehle analysiert und ausgeführt werden. Die verfügbaren Befehle sowie die etwaig benötigten externen Eingaben können in einer dafür vorgesehenen Datei definiert und anschließend vom Simulator verwendet werden. Nach

der Simulation wird eine Protokolldatei erstellt und die entsprechenden Informationen und Ergebnisse in dieser abgespeichert. Ein Ablaufplan des Simulators ist in Anhang D.2 unter Abbildung D.3 sowie ein Bild der Ausgabe des Simulators in Anhang D.2 zu finden.
Um sowohl die funktionelle Korrektheit, als auch die Qualität des Ergebnisses des MACG zu zeigen, wurden verschiedene Experimente durchgeführt.

Der Versuchsaufbau wird in Abbildung 5.15 dargestellt.

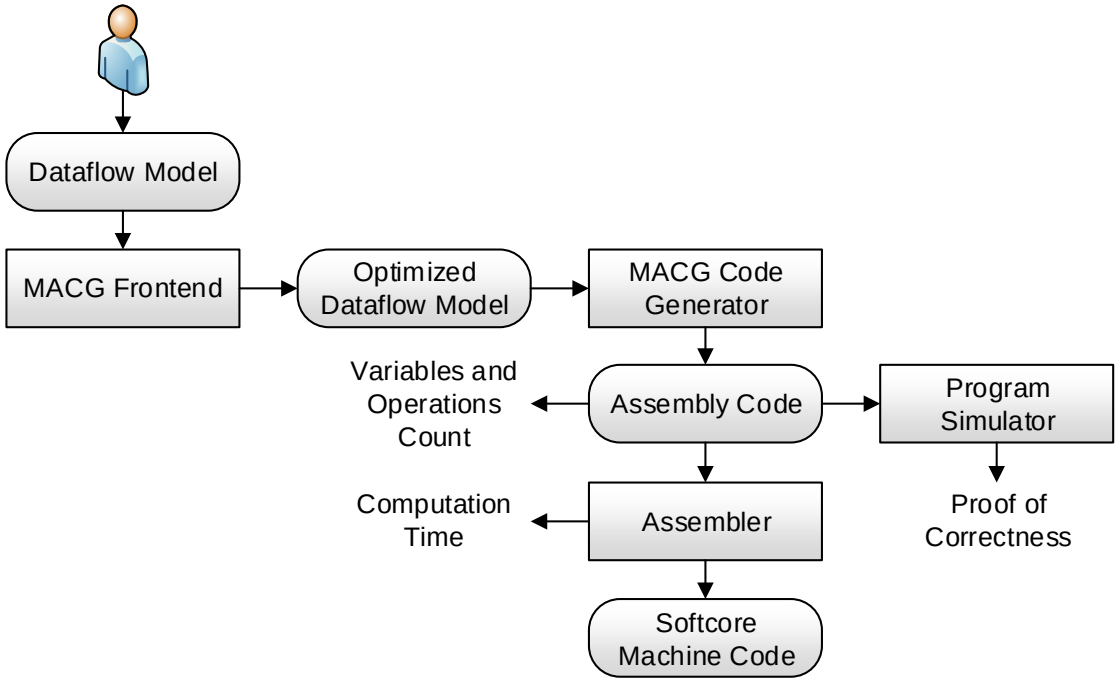



Abbildung 5.15: Versuchsaufbau MACG Experimente

Ein Nutzer hat mit Hilfe des MACG Frontend (*MACG Frontend*) alle Algorithmen als Datenflussgraphen (*Dataflow Model*) umgesetzt. Diese wurden dann in verschiedenen optimierten Konfigurationen (*Optimized Dataflow Model*) durch einen Codegenerator (*MACG Code Generator*) in Assemblercode sequentialisiert und (je nach Konfiguration) optimiert. Optimiert heisst in diesem Fall die Durchführung der Variablenreduktion. Der realisierte Assembler (*Assembler*) wurde zum übersetzen des Assemblercodes in Maschinencode (*Softcore Machine Code*) verwendet um die resultierende Rechenzeit (*Computation Time*) und die dadurch berechenbare Pipelineauslastung bestimmen zu können.

Die funktionelle Korrektheit des jeweilig resultierenden Assemblercodes wurde dabei anhand der Ergebnisse des eben vorgestellten Python Simulators im Vergleich mit funktionell gleichem, aber originalen, Assemblercode aufgezeigt. Hierzu wurden verschiedene Algorithmen exemplarisch ausgewählt und umgesetzt, die bereits in einem anderen Softcore Prozessor, dem LiSARD, zum Einsatz gekommen sind und unter [DPZ+13, DPF12] veröffentlicht wurden. Ein Vergleich mit einem anderen Softcore Prozessor der eine vergleichbare Aufgabendomäne hat, ist dabei wichtig um Aussagen bezüglich der Qualität der umgesetzten Optimierungen treffen zu können. Weiterhin wurden eigene Algorithmen umgesetzt und entsprechende Ergebnisse unter [KWS+18] ebenfalls veröffentlicht.

Anhand dieser Beispiele wurde ebenfalls die Qualität der entwickelten und umgesetzten Optimierungsalgorithmen gezeigt.
Es wurden alle bereits in Abschnitt 5.1.1 unter „Experimente zur Reduktion der Rechenzeit bei mehreren Kernen“ im Detail vorgestellten Algorithmen für diese Experimente umgesetzt und verwendet.
Zu diesen aus [DPZ+13,DPF12] entnommenen Assemblercodes ist zu sagen, dass, nach Aussage der Autoren, jeder dieser Assemblercodes auf die jeweilige Aufgabe hin optimiert wurde. Einige dieser Programme stammen aus praktischen Projekten und sind damit nachweislich qualitativ geprüft. Aus diesem Grund kann ein qualitativer Vergleich gezogen werden.
Im Folgenden werden die Ergebnisse vorgestellt:
Die Testeinstellungen wurden dabei so gewählt, dass zum einen alle Optimierungen deaktiviert wurden (MACG-Spalte) und in einem weiteren Durchlauf alle in Abschnitt 4.5.2 bis Abschnitt 4.5.4 vorgestellten Optimierungsverfahren aktiviert (MACG Opt-Spalte). Dabei wurde im Speziellen die Sequenzialisierungsstrategie Rollout Priority verwendet, da diese in Kombination mit der Variablenreduktion die besten Ergebnisse erzielt. Die detaillierten Konfigurationen, die während den Tests verwendet wurden, können im Anhang D.3 nachgelesen werden. Als Vergleichsgrundlage (Original-Spalte) dienen die im Rahmen des LiSARD [DPZ+13] erstellten Versionen dieser Algorithmen.
Die dargestellten Ergebnisse ergeben sich wie folgt:

- Anzahl an Variablen (*Variable Count*): abgelesen aus dem erzeugten Assemblercode
- Anzahl an Codezeilen (*Lines of Code Count*): abgelesen aus dem erzeugten Assemblercode
- Pipelineauslastung: Berechnung nach Gleichung 4.49
- Rechenzeit (*Computation Time*): Angabe des Assemblers nach dem Übersetzungsvorgang Assemblercode → Maschinencode durch einen eingebauten Taktzähler
- Korrektheit der Ergebnisse (*Result*): Simulation durch Python Simulator

| | Anzahl der Variablen | | |
|---|---|---|---|
| Name | Original | MACG | MACG Opt |
| FIR | 228 | 324 | 72 |
| Kalman 4 | 190 | 226 | 75 |
| Kalman 8 | 379 | 443 | 146 |
| Dreiachs Regler | 160 | 150 | 51 |
| Sechsachs Regler | 915 | 1238 | 446 |
| Polynom 12-3 | 152 | 137 | 37 |
| Polynom 12-6 | 302 | 281 | 85 |
| Polynom 24-6 | 532 | 557 | 145 |
| Polynom 48-6 | 990 | 1107 | 271 |
| Ellipse 1x | 282 | 391 | 75 |
| Ellipse 2x | 488 | 752 | 169 |

Tabelle 5.15: Anzahl an Variablen im Vergleich mit originalem Assemblercode

In Tabelle 5.15 sind die Ergebnisse der benötigten Anzahl an Variablen und damit Anzahl an minimal benötigten Speicherzellen der erzeugten Assemblercodes dargestellt. Hier werden die originalen Werte mit den durch den MACG erzeugten Assemblercode verglichen und

die durch die verschiedenen Optimierungsschritte erzielten Speicheroptimierungen werden zusammengefasst dargestellt. In der linken Spalte sind die in den originalen Versionen verwendeten Variablenanzahlen dargestellt. Die mittlere Spalte zeigt die benötigten Variablen ohne Optimierungen und die rechte Spalte die Anzahl an Variablen mit Optimierungen an.
Es ist anzumerken, dass auf dieser Abstraktionsebene (innerhalb des Assemblercodes) Variablen verwendet werden. Es findet an dieser Stelle noch kein physisches Mapping von Variablen auf Speicherzellen statt. Dieses Mapping wird erst bei dem Übersetzungsvorgang des hier erzeugten Assemblercodes in Maschinencode (durch den Assembler) durchgeführt.
Zur Auswertung der Ergebnisse gibt es zwei Ansatzpunkte:

- der Vergleich mit den Originalwerten
- der Vergleich mit ausgeschalteten Optimierungen

Im Folgenden werden die Werte des optimierten Ergebnisses (MACG Opt-Spalte) mit den durch die originalen Versionen erzielten Ergebnisse verglichen:

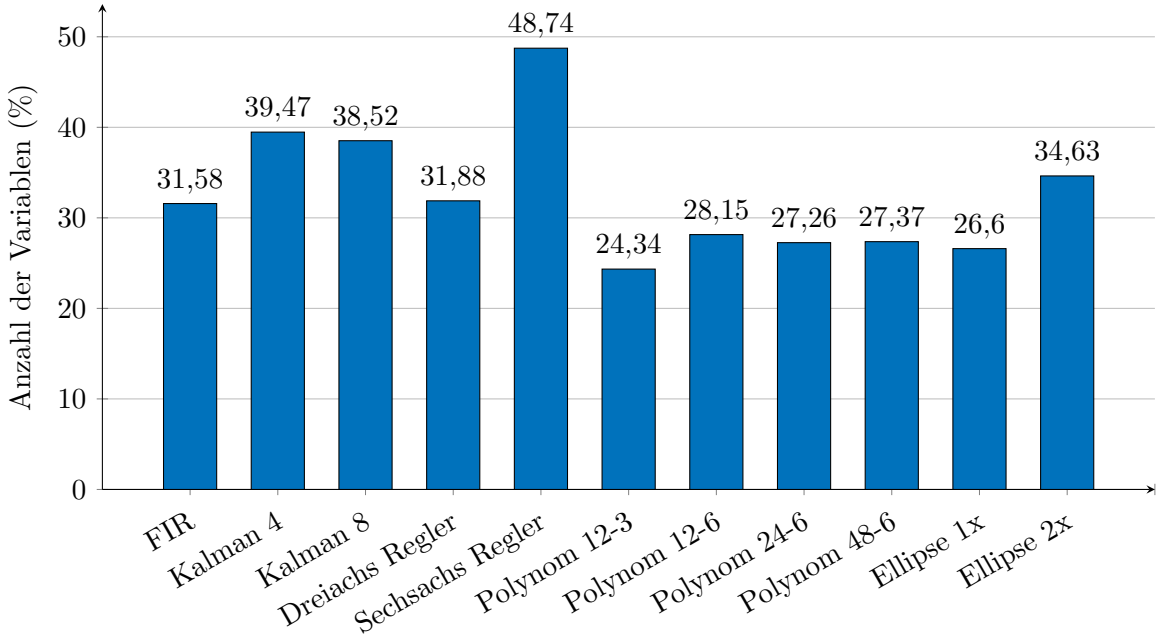



Abbildung 5.16: Anzahl an Variablen relativ zu hangeschriebenen Versionen

Zu sehen ist, dass die benötigte Anzahl an Variablen deutlich geringer als bei den originalen Versionen ausfällt. Das ist vermutlich dadurch zu erklären, dass die entsprechenden Assemblercodes so erstellt wurden, dass die einzelnen Befehle besonders gut parallelisierbar sind. Das wird erreicht indem vergleichsweise viele (teilweise redundante) Variablen verwendet werden. Um also eine bessere Aussage über die Qualität der erzielten Ergebnisse geben zu können, müssen weitere Parameter verglichen werden. Hierzu wird im Weiteren die Anzahl an Codezeilen des optimierten Assemblercodes (MACG Opt-Spalte) der resultierenden Assemblercodes mit den Ergebnissen der originalen Versionen verglichen. Eine Übersetzung durch den Assembler im Vorfeld des Vergleichs ist notwendig, da die Ergebnisse aufgrund von durch die Variablenreduktion entstandenen Abhängigkeiten verschieden ausfallen können. In einem weiteren Schritt werden alle Programme durch den in Abschnitt 5.3 vorgestellten Assembler übersetzt und auf die erzielte

Pipelineauslastung und die resultierende benötigte Rechenzeit miteinander verglichen.
Die Pipelineauslastung ist dabei definiert als eine prozentuale Angabe, in wievielen Takten jeweils eine neue Operation gestartet wird. Wird in jedem Takt eine neue Operation gestartet, so erreicht die Pipelineauslastung 100 %.

| | Anzahl der Codezeilen | | |
|---|---|---|---|
| Name | Original | MACG | MACG Opt |
| FIR | 198 | 196 | 81 |
| Kalman 4 | 175 | 176 | 170 |
| Kalman 8 | 350 | 352 | 340 |
| Dreiachs Regler | 108 | 108 | 97 |
| Sechsachs Regler | 882 | 958 | 890 |
| Polynom 12-3 | 112 | 101 | 77 |
| Polynom 12-6 | 223 | 209 | 161 |
| Polynom 24-6 | 445 | 413 | 297 |
| Polynom 48-6 | 889 | 821 | 450 |
| Ellipse 1x | 364 | 442 | 436 |
| Ellipse 2x | 728 | 878 | 872 |

Tabelle 5.16: Anzahl Codezeilen im Vergleich mit originalem Assemblercode

Die Ergebnisse des MACG sind vergleichbar gut, wie die Ergebnisse der originalen Versionen der Algorithmen, mit Ausnahme der Polynomberechnungen sowie des FIR-Filters. Hier weicht das Ergebnis stark von den originalen Version ab. Alle Assemblercodes wurden mittels des Python Simulators überprüft und errechnen trotz der schwankenden Anzahl an Codezeilen die gleichen Rechenergebnisse, rechnen also korrekt.
In Grafik 5.17 werden die Ergebnisse des resultierenden Maschinencodes der Optimierungen wieder in Relation zu den jeweiligen resultierenden Maschinencodes der originalen Versionen gestellt.

Um im Folgenden die Qualität der Ergebnisse einordnen zu können und damit den durch den modellbasierten Ansatz erzeugten enstehenden Abstraktionsoverhead (also den Overhead der aus dem höheren Abstraktionsgrad entsteht) beurteilen zu können, müssen sowohl die mit den jeweiligen Assemblercodes erreichten Pipelineauslastungen, als auch die vom Algorithmus im Softcore benötigte Rechenzeit berechnet werden. Hierzu wird der ebenfalls in dieser Arbeit entwickelte und im Abschnitt 5.3 vorgestellte Assembler zur Ermittlung der Pipelineauslastung verwendet. Da allerdings der LiSARD Softcore ein 10-Bit breites Adressfeld und damit bis zu 1.023 Speicherzellen verwendet, der ViSARD allerdings in der aktuellen Konfiguration nur ein 8-Bit Adressfeld und somit nur maximal 255 Speicherzellen verwendet, wird der Assembler zu Auswertungszwecken auf ein 10 Bit Adressfeld angepasst. Es werden im Weiteren lediglich die originalen Assemblercodes mit den optimierten MACG Assemblercodes verglichen.
Die Ergebnisse sind in Tabelle 5.17 dargestellt.

Wie in Tabelle 5.17 zu sehen ist, unterscheidet sich die durch den MACG Assemblercode erzielte Pipelineausnutzung teilweise stark zu den jeweiligen originalen Versionen. Das liegt allerdings daran, dass in den jeweiligen Beispielen, z. B. der FIR-Filter, der MACG Optimierer im Vorfeld redundante oder unnötig komplexe Operationen ersetzt hat, was an der resultierenden Verkürzung der Länge des Assemblercodes um 59,09 % gesehen werden kann. Hierbei wurde

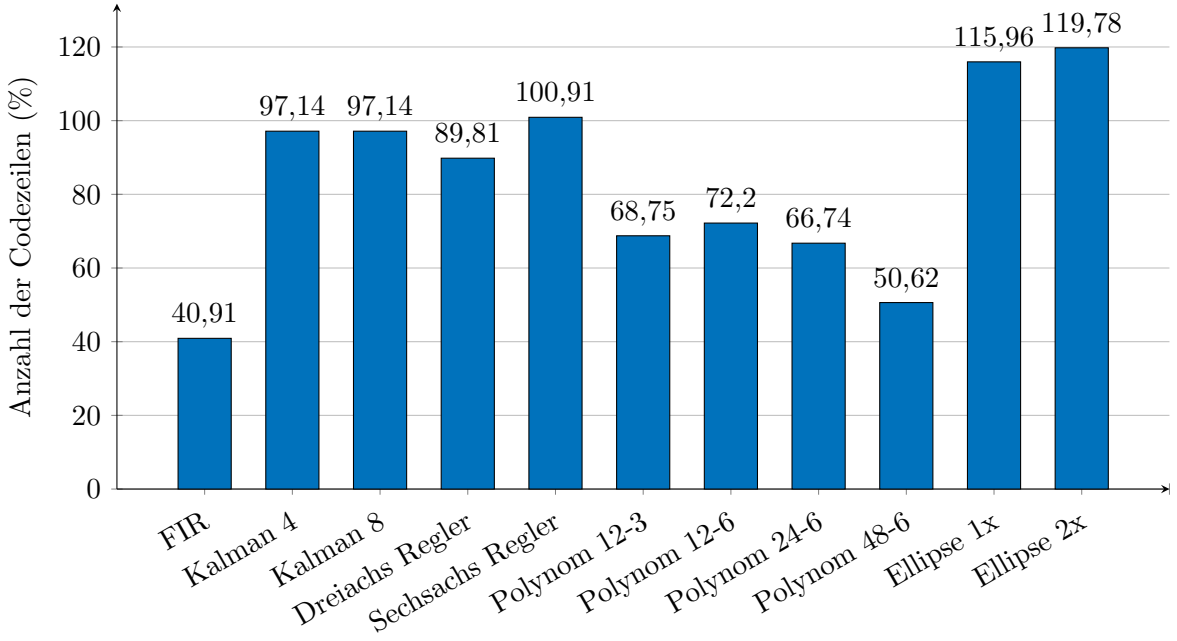



Abbildung 5.17: Anzahl an Codezeilen des optimierten Ergebnisses relativ zu den originalen Versionen

| | Pipelineauslastung in % | | Rechenzeit in ns | |
|---|---|---|---|---|
| Name | Original | MACG Opt | Original | MACG Opt |
| FIR | 83,1 | 49,0 | 2.360 | 1.520 |
| Kalman 4 | 61,0 | 59,6 | 2.870 | 2.850 |
| Kalman 8 | 79,7 | 77,8 | 4.390 | 4.370 |
| Dreiachs Regler | 48,6 | 44,9 | 2.220 | 2.160 |
| Sechsachs Regler | 88,2 | 89,8 | 9.820 | 9.910 |
| Polynom 12-3 | 68,7 | 41,4 | 1.630 | 1.860 |
| Polynom 12-6 | 82,3 | 68,5 | 2710 | 2350 |
| Polynom 24-6 | 89,5 | 81,6 | 4.970 | 3.590 |
| Polynom 48-6 | 94,7 | 89,0 | 9.390 | 6.260 |
| Ellipse 1x | 28,7 | 29,0 | 14.160 | 15.020 |
| Ellipse 2x | 45,9 | 49,7 | 15.850 | 17.560 |

Tabelle 5.17: Pipelineauslastung und Rechenzeit Original vs. MACG optimiert

zwar die effektive Pipelineausnutzung von 83,1 % auf 49,0 % , also um 34,1 % verringert, die Rechenzeit des Algorithmus im Softcore hat sich allerdings von 2.360 ns auf 1.520 ns verringert. Ein weiteres Beispiel, Polynom 12-3, erzielt eine resultierende Verkürzung der Länge des Assemblercodes um 31.25 %. Hierbei wurde zwar die effektive Pipelineausnutzung von 68.7 % auf 41.4 %, also um 27.3 %, verringert, die Rechenzeit des Algorithmus im Softcore hat sich allerdings von 1.630 ns auf 1.860 ns, also um 14.11 %, erhöht. Der Overhead, der aus dem höheren Abstraktionsgrad entsteht, beträgt für das Beispiel Polynom 12-3 also 14.11 %.

Aus diesen Werten lässt sich für alle hier getesteten Beispiele ein durchschnittlicher Overhead, der aus dem höheren Abstraktionsgrad entsteht, von -7,45 % berechnen. Das bedeutet, dass

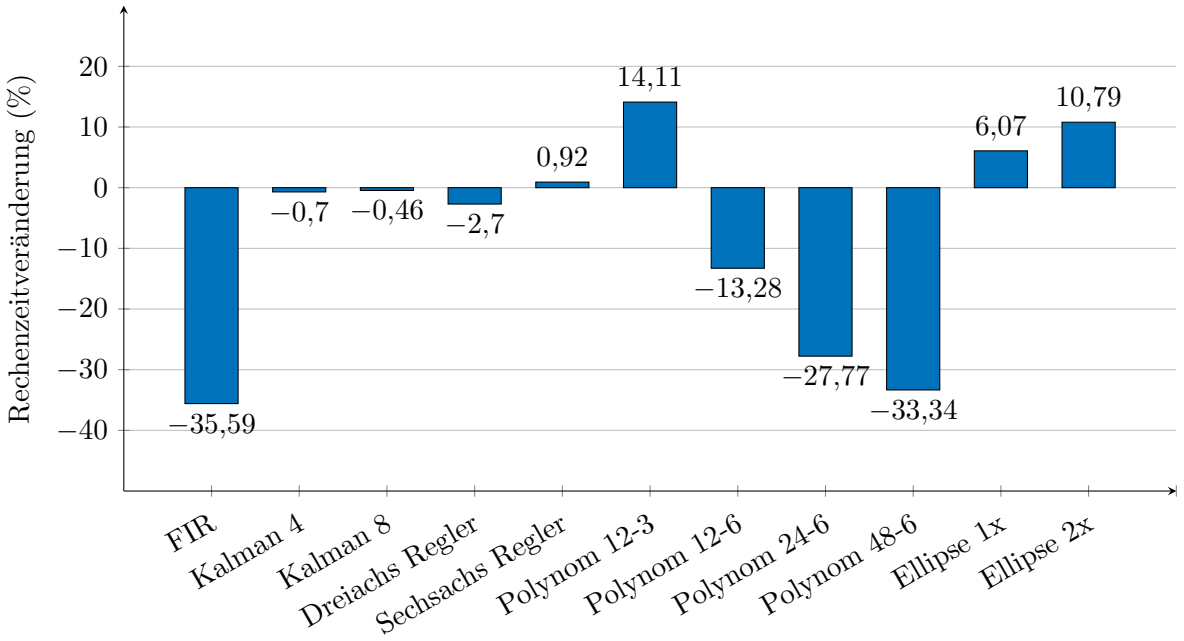



Abbildung 5.18: Rechenzeitveränderung relativ zu hangeschriebenen Versionen

im Durchschnitt die durch den MACG erstellten und optimierten Assemblercodes 7,45 % schneller sind als originalen Versionen. Dabei benötigen diese im Durchschnitt nur 32,59 % der Speicherzellen im Vergleich zu den originalen Versionen. Es ist also möglich bei den hier getestetem Assemblerprogrammen über zwei Drittel des benötigten Speichers einzusparen bei gleichzeitiger Beschleunigung der Abarbeitungsgeschwindigkeit von 7,45 %.

#### Vergleich der erzeugten Assemblercodes mit und ohne Variablenreduktion

Da im Rahmen der MACG-Optimierungen die im Assemblercode verwendeten Variablen minimiert werden können, wie in Abschnitt 4.5.4 vorgestellt, ergibt sich daraus die Frage, ob die dadurch zusätzlich entstandenen WaR- und WaW-Abhängigkeiten einen negativen Einfluss auf die resultierende Rechenzeiten der Maschinencodes im Softcore haben. Die Reduzierung der Anzahl an benötigten Variablen wurde bereits in Tabelle 5.15 (MACG und MACG-Opt Spalten) vorgestellt. Um diesen Einfluss zu testen, wurden sämtliche bisher durch den MACG erstellten Assemblercodes durch den Assembler übersetzt und die resultierende Rechenzeit miteinander verglichen. Hierzu wurden die in Anhang D.3 vorgestellten Testkonfigurationen verwendet. Dabei wurden die Assemblercodes mit Variablenreduktion (MACG Opt) mit den Assemblercodes ohne diese Optimierung (MACG) vom Assembler in zwei Varianten übersetzt:

- Mit ausgeschalteter Assembler-Speicherverwaltung (siehe Abschnitt 4.6.2)
  Die Variablen im Assemblercode erhalten ein 1:1-mapping im Speicher, jede Variable wird fest auf eine Speicherzelle gelegt

- Mit eingeschalteter Assembler-Speicherverwaltung
  Die Variablen im Assemblercode haben ein 1:n-mapping im Speicher, jede Variable kann in unterschiedlichen Werten auf unterschiedlichen Speicherzellen abgespeichert werden.

Die Ergebnisse sind in Tabelle 5.18 dargestellt.

| | Rechenzeit in ns | | | |
|---|---|---|---|---|
| | Ohne Speicherverwaltung | | Mit Speicherverwaltung | |
| Name | MACG | MACG Opt | MACG | MACG Opt |
| FIR | 1.520 | 3.060 | 1.520 | 1.520 |
| Kalman 4 | 4.270 | 4.760 | 2.850 | 2.850 |
| Kalman 8 | 6.780 | 9.310 | 4.370 | 4.370 |
| Dreiachs Regler | 2.160 | 3.020 | 2.160 | 2.160 |
| Sechsachs Regler | - | 21.390 | 9.910 | 9.910 |
| Polynom 12-3 | 1.860 | 2.440 | 1.860 | 1.860 |
| Polynom 12-6 | 2.350 | 5.020 | 2.350 | 2.350 |
| Polynom 24-6 | 3.590 | 8.110 | 3.590 | 3.590 |
| Polynom 48-6 | 6.290 | 12.180 | 6.260 | 6.260 |
| Ellipse 1x | 15.020 | 15.020 | 15.020 | 15.020 |
| Ellipse 2x | 18.310 | 22.660 | 17.560 | 17.560 |

Tabelle 5.18: Resultierende Rechenzeit mit und ohne Variablenreduktion

Wie in Tabelle 5.18 gesehen werden kann, ist die Variablenreduktion des MACG kein Nachteil, sofern der nachfolgende Assembler eine Speicherverwaltung realisiert, die es ermöglicht die gleiche Variable auf mehrere Speicherzellen zu verteilen. Die Ergebnisse sind in beiden Fällen identisch.
Hat der Assembler allerdings eine feste Zuordnung von Variable und Speicherzelle, so erhöht sich die Rechenzeit der Assemblerprogramme, wenn eine Variablenreduktion verwendet wurde. Dass die Variablenreduktion auch in diesem Fall sinnvoll ist, zeigt das Beispiel des Sechachs Reglers. Wie der Tabelle 5.18 entnommen werden kann, kann dieser Assemblercode nicht durch den Assembler übersetzt werden, wenn keine Variablenreduktion vorgenommen wurde. Der Grund dafür liegt in der steigenden Anzahl an Variablen mit steigender Codelänge des Assemblercodes. Jeder neue Befehl erzeugt potenziell eine neue Variable. Das bedeutet, bei langen Assemblercodes werden mehr Variablen benötigt, als der Softcore im Speicher adressieren kann. Im Fall dieses Tests sind das 1.023 ($2^{10}$-1) Variablen, der Sechachs Regler benötigt allerdings in dieser Konfiguration bereits 1.238 Variablen. Lange Assemblercodes führen also unweigerlich zu einem sehr großen notwendigen Speicher, der wiederum FPGA-Ressourcen benötigt. Auf diesem Grund ist es bei langen Assemblercodes sinnvoll, auch bei einer festen Variablen-Speicherzelle-Zuordnung, die Variablenreduktion zu verwenden, da der Assemblercode dann zwar viel Rechenzeit auf dem Softcore benötigt, aber ansonsten nicht mit dem Softcore abgearbeitet werden kann. Ein zusätzlicher Faktor ist, dass der verfügbare angenommene Speicher des Softcores für dieses Experiment mit 1023 Speicherzellen festgelegt worde, um der Anzahl an Speicherzellen des LiSARD zu entsprechen. Diese Anpassung war notwendig, um eine gemeinsame Vergleichsgrundlage zu schaffen. Der in dieser Arbeit adressierte Softcore verwendet im Normalfall lediglich 255 Speicherzellen. Mit dieser Einschränkung könnten ohne eine Speicherverwaltung die Algorithmen FIR, Kalman 8, Sechachs Regler, Ellipse 1x und 2x, sowie die Polynome 12-6, 24-6 und 48-6 nicht ausgeführt werden, da diese zu viele Speicherzellen für die verwendeten Variablen benötigen würden (Vergleich mit Tabelle 5.15). Das würde bedeuten, dass ohne die umgesetzte Speicherverwaltung und mit 255 Speicherzellen im Softcore acht der elf getesteten Assemblerprogramme nicht verwendet werden könnten. Die Konfiguration mit 255 Speicherzellen ist allerdings in vielen eingebetteten System sinnvoll, um die Anzahl an

benötigten Ressourcen im FPGA gering zu halten und nicht unnötig viele Ressourcen für einen Speicher zu verwenden, der lediglich zu Programmstart effizient und anschließend nur noch zu einem Bruchteil genutzt wird.
Ein weiterer Punkt ist die Tatsache, dass die Einhaltung der harten Echtzeitschranke das primäre Kriterium ist. Dabei ist es egal, ob diese Schranke exakt erreicht oder deutlich unterschritten wird. Das bedeutet, sobald das Kriterium erfüllt ist, bringt eine weitere Erhöhung der Rechengeschwindigkeit keinen Vorteil. Mit Hilfe der Variablenreduktion kann allerdings Speicher, also Ressourcen auf dem FPGA, eingespart werden. Hier hat jede weitere Einsparung einen positiven Effekt, sofern dadurch ein kleinerer Speicher verwendet und damit Ressourcen eingespart werden können.

#### Experiment mit einem echtzeitkritischen Bildverarbeitungsalgorithmus

Im Weiteren wurde ein sehr komplexer Algorithmus zur Bildverarbeitung umgesetzt, der allerdings nicht auf anderen Softcore Prozessoren zum Einsatz gekommen ist und deswegen separiert betrachtet wird. Es handelt sich dabei um einen Algorithmus zur Weißlichtinterferometriedatenauswertung.
Weißlichtinterferometrie (*White Light Interferometry* (*WLI*)) ist ein Verfahren, bei dem mittels eines Michelson Interferometers eine Oberfläche mit einem breiten Spektrum an Licht bestrahlt und entsprechende Bilder aufgenommen werden, während der Abstand zum Messobjekt vergrößert wird. Für jeden Messpunkt ergibt sich ein Interferenzpattern durch die Pixel des gleichen Punktes der unterschiedlichen Abstände. Die Änderung der Intensität in Korrelation mit der Abstandsvergrößerung wird als Korrelogramm bezeichnet. [MMF14]
Diese Algorithmus stellt einen typischen Bildverarbeitungsalgorithmus im Sinn der Eingrenzung des Aufgabenspektrums des Softcore Prozessors dar, wie in Abschnitt 2.6 vorgestellt. Gleichzeitig hat dieser Algorithmus eine starke praktische Relevanz in den im Rahmen der Dissertation bearbeiteten Forschungsprojekte.
Der Algorithmus benötigt Korrelogramme in Form von Grauwerten als Eingabeargumente und erstellt und glättet eine Höhenkurve um die genauen Positionen etwaiger Maxima, also beispielsweise von Kanten, in jedem Korrelogramm zu finden.

Dabei wurden zwei Versionen umgesetzt: Version Eins mit 64 Stützstellen und Version Zwei mit 28 Stützstellen. Bei der zweiten Version sind diese Stützstellen bereits vorverarbeitet. [KWS+18]
Exemplarisch wurden für diese Algorithmen alle verfügbaren Sequenzialisierungsverfahren in Kombination mit den Optimierungsverfahren eingesetzt, um aufzuzeigen, dass die Rollout Priority Sequenzialisierungsstrategie das beste Ergebnis liefert.
Hierzu wurden alle Sequentialisierungsstrategien, die in Abschnitt 4.5.3 vorgestellt wurden, mit anschließender Variablenoptimierung verwendet. Die Zeile MACG FiFo entspricht dabei der FiFo Sequentialisierungsstrategie, MACG Prio der prioritätsbasierten Sequentialisierungsstrategie und MACG Opt der Rollout Priority Sequentialisierungsstrategie.

Wie der Tabelle 5.19 entnommen werden kann, ist die Rollout Priority die Sequenzialisierungsstrategie, die das beste Ergebnis im Punkt Speicherverringerung erzielt. Nur mit dieser Strategie ist es überhaupt möglich, die erste Version des WLI-Algorithmus mit dem Softcore Prozessor und einer 8 Bit-Konfiguration auszuführen. Selbst der Vergleichssoftcore LiSARD mit einer 10 Bit breiten Speicheradressierung besitzt nicht genügend reservierten Speicher um den Algorithmus mit einer anderen Konfiguration als der Rollout Priority ausführen zu können. Aus diesem Grund sind in Tabelle 5.19 in den Reihen MACG FiFo und MACG Prio keine

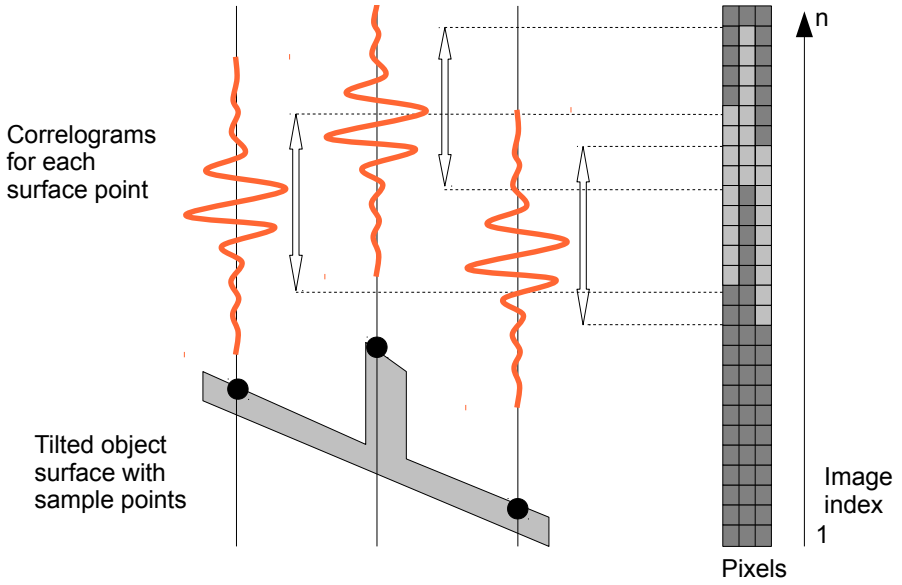



Abbildung 5.19: WLI Korrelogramme im Bildstapel (aus [MMF14])

| WLI-Algorithmus | | Anzahl an Variablen | Anzahl an Codezeilen | Pipelinenutzung (%) | Rechenzeit (ns) |
|---|---|---|---|---|---|
| V1 | Orig | 99 | 34.011 | 69,75 | 487.613 |
| | MACG | 28.913 | 34.026 | - | - |
| | MACG FiFo | 1.486 | 34.021 | - | - |
| | MACG Prio | 1.421 | 34.021 | - | - |
| | MACG Opt | 79 | 34.021 | 68,51 | 496.584 |
| V2 | Orig | 58 | 1.075 | 61,6 | 16.180 |
| | MACG | 1.040 | 1.095 | 61,6 | 16.180 |
| | MACG FiFo | 178 | 1.068 | 66,0 | 16.180 |
| | MACG Prio | 154 | 1.068 | 66,0 | 16.190 |
| | MACG Opt | 99 | 1.070 | 66,2 | 16.140 |

Tabelle 5.19: Ergebnisse der WLI-Algorithmen

Werte bei Pipelineausnutzung und Rechenzeit angegeben. Mit dem für diese Tests verwendeten Assembler ist es zwar möglich, verschiedene Variablen auf ein und dieselbe Speicherzelle zu mappen, aber im MACG werden die meisten Variablen mit einem Wert initialisiert und werden gelesen, bevor diese geschrieben werden. Bei diesem Szenario müssen zu Programmstart alle Variablen also in zunächst unterschiedliche Speicherzellen geschrieben werden. Da aber mehr Variablen verwendet werden, als Variablenspeicher verfügbar ist, kann für diese Konfigurationen kein korrektes Scheduling gefunden werden. Die Anzahl an verwendeten Variablen wird in der Grafik 5.20 prozentual zur nicht optimierten Version (MACG-Zeile) dargestellt.

### 5.2.4 Fazit

Mit Hilfe des MACG wurde ein Matlab/Simulink Plugin erstellt, das es dem Nutzer ermöglicht, auf modellbasierter Ebene beliebige Assemblercodes für den Softcore Prozessor ViSARD erstellen

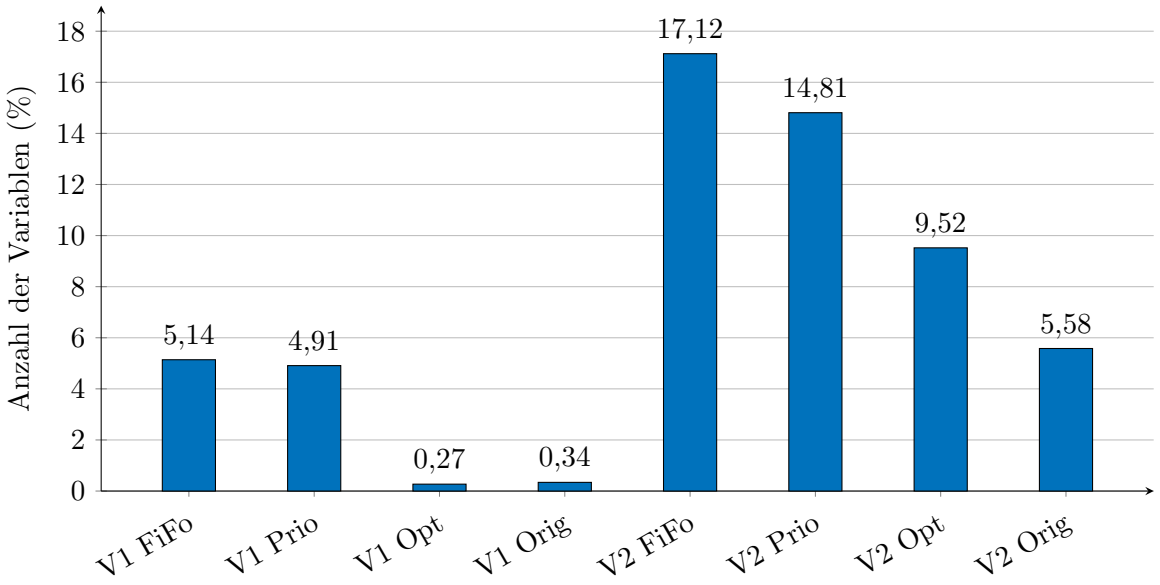



Abbildung 5.20: Anzahl an Variablen relativ zu nicht optimierten Versionen

zu können. Hierbei muss der Nutzer keine Kenntnisse über die Architektur des Softcore, oder über den genauen Aufbau, also die Syntax, des zugrunde liegenden Assemblercodes besitzen. Es genügt das gestellte Problem mittels eines Datenflussmodells nachzubilden. Das Tool übersetzt das Datenflussmodell anschließend voll automatisiert zu einem funktionskorrekten Assemblercode und optimiert in verschiedenen Zwischenschritten zunächst den erstellten Graphen. In einem weiteren Schritt wird mittels eines zuvor gewählten Sequenzialisierungsverfahrens der Graph sequenzialisiert. Hierbei wurde durch Experimente mit verschiedenen Algorithmen gezeigt, dass entwickelte Rollout Priority die besten Voraussetzungen bietet, um in einem letzten Optimierungsschritt den benötigten Variablenspeicher jedes Assemblercodes zu minimieren. In weiteren Experimenten wird gezeigt, welchen Einfluss die Reduktion der im Assemblercode verwendeten Variablen auf die resultierende Maschinencode Rechenzeit hat.
Mit dem MACG konnte gezeigt werden, dass alle entwickelten und entsprechend kombinierten Methodiken sowohl zur Optimierung von gegebenen Multi-Graphen, wie im Methodik-Abschnitt unter 4.5.2 gezeigt, als auch die entwickelten Sequenzialisierungsstrategien und der darauf aufbauende Speicherreduzierungsalgorithmen, funktionieren.
Das Resultat ist ein optimierter Assemblercode, dessen Abstraktionsoverhead in einem akzeptablen Rahmen gegenüber vergleichbaren, funktionsgleichen handgeschriebenen Assemblercodes bietet. In vielen Fällen ist der durch den MACG erstellte Assemblercode sogar effizienter sowohl in resultierender Softcore Abarbeitungsgeschwindigkeit, als auch bei den benötigten Speicherressourcen der Variablen und Konstanten.
Sämtliche im Abschnitt 5.2.3 durchgeführten Experimente entsprechen dem aus Abbildung 5.1 vorgestellten Teil des MACG unter zusätzlichem Einsatz des optionalem Programmsimulators (*Program Simulator*). Mit diesen Experimenten konnte unter anderem die Erstellung, Optimierung und Evaluation von Assemblercode über ein modellbasiertes Tool, dem MACG, nachgewiesen werden. Es wurde gezeigt, dass es möglich ist damit einen effizienten Assemblercode zu erzeugen. Potenzielle Probleme wie die Übergabeschnittstelle zwischen Assembler und MACG in Form von einer zu hohen Speicherbelegung durch den erzeugten Assemblercode wurden beleuchtet und eine entsprechende Lösung ausgewertet.

Zum Zeitpunkt dieser Arbeit besteht der MACG aus exakt den Blöcken, die eineindeutig Assemblerbefehlen zugeordnet werden können und einem Schleifenblock. Der Nutzer des Tool muss dadurch allerdings beim Erstellen der Datenflussgraphen bestimmte Annahmen zur Reihenfolge kommutativer Operationen treffen und dadurch kann ein Assemblercode entstehen, der innerhalb dieser Operationen nicht Rechenzeitoptimiert ist. Da eine optimale Reihenfolge allerdings automatisiert festgelegt werden kann, sollte das Tool um Blöcke mit variablen Anzahl an Argumenten für alle kommutativen Operationen erweitert werden. Ein Algorithmus kann beim Übersetzungsvorgang dann eine Rechenzeitoptimierte Sequentialisierung festlegen und umsetzen.

## 5.3 Optimierender Softcore-Assembler

Der ViSARD Assembler ist ein Kommandozeilen basiertes, eigenständiges Tool, mit dem es möglich ist, automatisiert jeden gegebenen und nach Anhang B aufgebauten Assemblercode in für den ViSARD lesbaren Maschinencode zu übersetzen. Der Assembler ist in der Programmiersprache C# umgesetzt worden, da diese vorteilhafte Eigenschaften wie beispielsweise eine Garbage-Collection und Objekt-orientierte Programmierung bietet.
Die in dieser Arbeit adressierte Problemklasse hat als wichtige Voraussetzung die Sicherstellung einer harten Echtzeitfähigkeit des resultierenden Maschinencodes. Diese Voraussetzung wird durch die Eigenschaften, den Aufbau sowie den Funktionsumfang des zugrunde liegenden Assemblercodes sichergestellt. Da eben zur Sicherstellung dieser Voraussetzung ein eigener Assemblercode entwickelt werden musste, ist ebenfalls ein Assembler notwendig, der diesen in Maschinencode übersetzen kann.
Bei dem Übersetzungsvorgang des Assemblercodes in Maschinencode können verschiedene Optimierungsstrategien zum Einsatz kommen, sodass der resultierende Maschinencode eine hohe Pipelineausnutzung hat, wie im weiteren Verlauf des Abschnitts über entsprechende Benchmarks gezeigt werden wird. Diese Optimierungsstrategien bilden einen wichtigen Teil des Assemblers und werden entsprechend in Teilabschnitt 5.3.4 experimentell untersucht. Die Maximierung der Verarbeitungsgeschwindigkeit ist also als Ziel des Assemblers definiert.
Dieser erzeugt dabei einen zur Designzeit voll analysierbaren Maschinencode, womit die garantierte Einhaltung der Echtzeitkriterien sichergestellt werden kann. Der Assembler ist dabei in der Lage, eine oder mehrere Quelltextdateien als Eingabe zu erhalten und diese in exakt zwei Maschinencodedateien zu übersetzen. Diese Funktion wird in Abbildung 5.21 illustriert.

Diese zwei Dateien beinhalten zum einen die Maschinencodebefehle (*Softcore Instruction Code*) für den Softcore und zum anderen Anweisungen, mit welchen Startwerten die Speicherzellen zu belegen sind (*Softcore Data Code*). Weiterhin besteht die Möglichkeit, den Assembler, bzw. genauer gesagt die Art des Schedulings und damit verbunden die Optimierungsverfahren des Assemblers, im Rahmen eines Projektes auf spezielle Eigenschaften des als Eingabe empfangenen Assemblercodes hin zu optimieren. Das ist mittels einer anpassbaren Assembler und Prozessorkonfiguration (*Processor + Assembler Configuration*) zu erreichen. Die notwendigen Informationen, wie diese Konfigurationen zu anzupassen sind, können über die durch den Assembler während eines Übersetzungsvorgangs erzeugten Statistiken und Informationen (*Informations + Statistics*) gewonnen werden. Um daraus zuverlässige Informationen zu erhalten, ist es sinnvoll eine Vielzahl an Versuchen mit unterschiedlichen Assemblercodedateien (*Assembly Source Code*) durchzuführen, bevor der eigentliche zu verwendende Assemblercode übersetzt wird. Diese Dateien können mittels eines

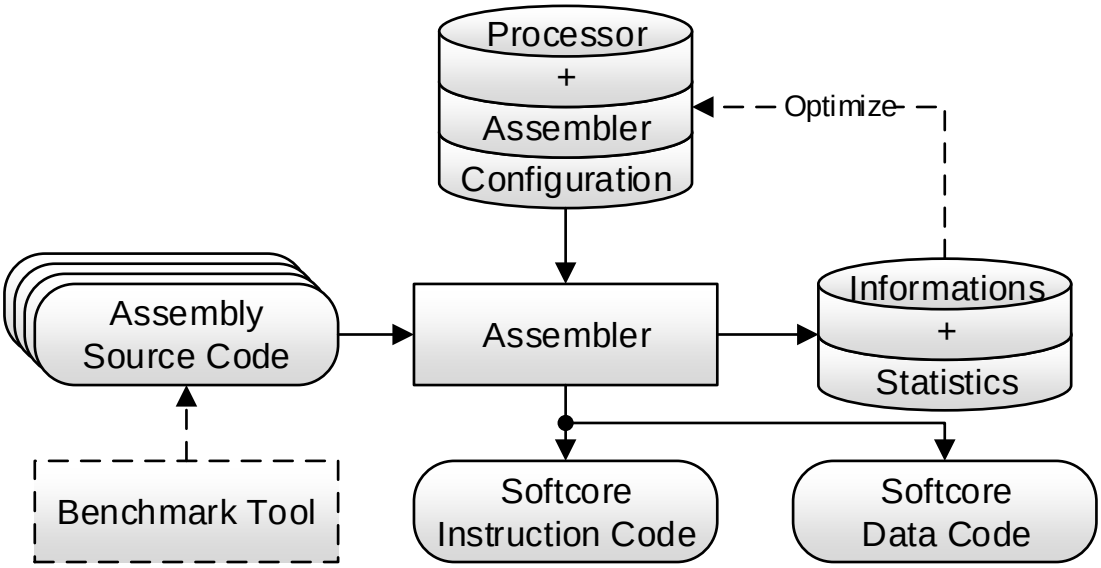



Abbildung 5.21: Ablauf einer Assembleriteration

speziell dafür erstellten Benchmark Tools (*Benchmark Tool*) automatisiert erzeugt und vom Assembler übersetzt werden lassen.

### 5.3.1 Das Benchmark-Tool

Im Folgenden wird das Benchmark Tool kurz vorgestellt und dessen Sinn und Funktionsweise erläutert. Das Benchmark-Tool mit Funktionsumfang und Eigenschaften wurden methodisch für die Dissertation definiert; eine praktische Umsetzung erfolgte im Rahmen der Bachelorarbeit von Herrn Kreuzkamp [Kre18]. Allgemein kann gesagt werden, dass das Tool nach Vorgabe des Nutzers (pseudo-)zufälligen Assemblercode erstellt und diesen automatisiert durch den Assembler übersetzen lässt. Mit den generierten Ergebnissen werden Statistiken erstellt, die ausgewertet werden können um die Optimierungseinstellungen des Assemblers zu verändern, oder gänzlich neue Optimierungsstrategien umzusetzen.
Innerhalb des Tools sind eine Reihe von Einstellungen auswählbar, die eine individuelle Anpassung des erzeugten Codes an die Eigenschaften des jeweiligen Projektes erlauben. Es können durchschnittlicher Abhängigkeitsgrad, Abhängigkeitsgrad der jeweiligen Typen (RaR, RaW, WaR und WaW, siehe Tabelle 4.4), Verteilungswahrscheinlichkeit der verfügbaren Operatoren, maximale Variablenzahl, Anzahl der Prozessorkerne (des ViSARD) sowie Codelängen und Wiederholungszahlen definiert werden. Weiterhin können die erstellten Assemblerdateien gespeichert, oder gespeicherte Dateien importiert und wiederverwendet werden.
Der Abhängigkeitsgrad einer Assembleranweisung $cmd_i$ ist definiert als der Wert $k$, wenn $k-1$ der maximale Abhängigkeitsgrad der Wertzuweisung an den jeweiligen Operanden ist, wobei nur Wertzuweisungen betrachtet werden, die vor $cmd_i$ ausgeführt werden. Wenn ein Operand noch keine Wertzuweisung bekommen hat, wird ein Abhängigkeitsgrad von 0 zugewiesen. [Wal74]
Konkret bedeutet das im erzeugten Assemblercode, da innerhalb jedes Befehls bis zu zwei

Argumente (*readOperand*) und ein Ergebnis (*writeOperand*) auftreten können, dass der jeweilige Assemblerbefehl den Abhängigkeitsgrad $Dep(i)$ (*Degree of Dependence*) des Ergebnisses annimmt, wie in Gleichung 5.2 gezeigt.

$$Dep(writeOperand_i) = max\{Dep(readOperand1_i), Dep(readOperand2_i)\} + 1 \tag{5.2}$$

Der Abhängigkeitsgrad des gesamten Assemblercodes ist also abhängig vom Durchschnitt der jeweiligen Befehle und damit von den gewählten Argumenten dieser Befehle. Das soll an folgendem Beispiel und der zugehörigen Abbildung 5.22 verdeutlicht werden:

$$F = A + B;$$
$$G = C + D;$$
$$H = C + F;$$
$$K = E + H;$$

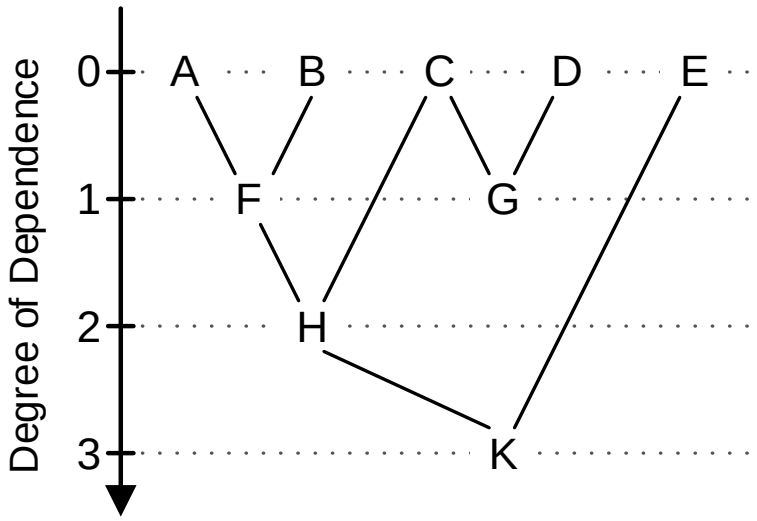



Abbildung 5.22: Beispielberechnung Abhängigkeitsgrad

Nach diesem Verfahren wird jedem Befehl (stellvertretend durch die Ergebnisvariable) ein Abhängigkeitsgrad zugeordnet. Die Abhängigkeitsgrade der Variablen $A$ bis $E$ werden mit 0 initialisiert, da diese bisher nicht als Ergebnis einer Operation verwendet wurden. $F$ und $G$ sind Ergebnisse der ersten zwei Additionen und erhalten damit den Abhängigkeitsgrad Eins. Dem Schema folgend, erhält $H$ den Abhängigkeitsgrad Zwei und $K$ den Abhängigkeitsgrad Drei, wie in Gleichung 5.2 festgelegt. Der durchschnittliche Abhängigkeitsgrad des gesamten Programms ergibt sich aus den Abhängigkeitsgraden aller Ergebnisse und ist für dieses Beispiel 1,75.

$$(1 + 1 + 2 + 3) \div 4 = 1,75$$

Der Abhängigkeitsgrad des gesamten (bisher erzeugten) Programms wird also aus dem Durchschnitt aller einzelnen Abhängigkeitsgrade gebildet.

Das bedeutet, bei jedem Befehl müssen Operanden gewählt werden, bei deren Verwendung der geforderte durchschnittliche Abhängigkeitsgrad erreicht wird. Allerdings darf nicht in jedem Fall der bestmögliche Operand gewählt werden, da die resultierenden Assemblercodes sich ansonsten zu sehr ähneln. Aus diesen Gründen wird bei der Wahrscheinlichkeitsberechnung eine Poisson-Verteilung [Kar14] verwendet, da hier eine Entscheidung, welche Varianz gewählt werden muss, entfällt, da die Funktion lediglich vom Erwartungswert abhängt.
Die Entscheidung, welcher Mnemonic für einen neuen Befehl ausgewählt wird, wird dabei über eine feste Wahrscheinlichkeit gewählt. Diese kann entweder durch den Nutzer definiert werden, alternativ weist das Tool jedem Mnemonic die gleiche Wahrscheinlichkeit von $\frac{1}{\#Mnemonic}$ zu.
Sobald das Mnemonic gewählt ist, kann anhand dessen bestimmt werden, welche der drei Operandenplätze mit Operanden, Portangaben bzw. Delimiter-Zeichen belegt werden müssen. Die Belegung mit Variablen als Operanden wird dabei entsprechend so gewählt, dass die vorgegebenen Abhängigkeitsgrade erfüllt werden. Der Abhängigkeitsgrad des Delimiter-Zeichens, sowie etwaiger Port-Angaben wird immer auf 0 definiert.

### 5.3.2 Spezifika des Softcore-Assemblers

Die wichtigste Aufgabe des Assemblers ist das Befehlsscheduling. Durch ein gutes Befehlsscheduling wird eine Rechenzeitoptimierung des resultierenden Maschinencodes erreicht. Da der ViSARD über verschiedene Pipelinestufen verfügt, kann die Abarbeitung einzelner Assemblerbefehle durch die Nutzung der jeweiligen Pipelines parallelisiert werden. Der ViSARD verfügt dabei über eine fünfstufige Befehlspipeline, außerdem ist jeder Operator intern mit einer weiteren Pipeline realisiert. Der genaue Aufbau des Softcores samt Diskussion der Pipeline wurde in Abschnitt 5.1 im Detail vorgestellt.
Die Aufgabe des Assemblers ist es also, die Reihenfolge der Operationen im Assemblercode so zu schedulen, dass die Pipelineauslastung im resultierenden Maschinencode maximiert wird. Dabei müssen sowohl die Abhängigkeiten der einzelnen Operationen zueinander (RaW,WaR und WaW, siehe Abschnitt 4.6.1 unter Tabelle 4.4 und zugehöriger Erläuterung) beachtet werden, als auch die unterschiedlichen Rechenzeiten der jeweiligen Operatoren. Die Rechenzeiten der Operatoren sind deswegen von Interesse, da aufgrund der dem ViSARD zugrunde liegenden Architektur in jedem Takt maximal ein Ergebnis in den Speicher geschrieben werden kann. Durch unterschiedliche Rechenzeiten der Operatoren kann es zu einer Situation kommen, bei der in einem Takt zwei (oder mehr) Operatoren ein Ergebnis erzeugen. Dieser Fall muss durch den Assembler verhindert werden. Die Beachtung der Abhängigkeiten stellt sicher, dass der im Assemblercode realisierte Algorithmus korrekt ausgeführt wird. Dabei können die WaR- und WaW-Abhängigkeiten durch Register Renaming aufgelöst werden, wie in Abschnitt 4.6.2 der Methodologie des Assemblers beschrieben. Die verschiedenen umgesetzten Optimierungsverfahren werden im folgenden Abschnitt 5.3.3 im Detail vorgestellt.
Der Assembler kann den gegebenen Assemblercode auf aktuell zwei verschiedene Genauigkeiten übersetzen, je nach Konfiguration des ViSARD. Hier kann zwischen einfacher floating-point Genauigkeit (Single Precision, 32 Bit) und doppelter floating-point Genauigkeit (Double Precision, 64 Bit) gewählt werden.
Der Aufbau des resultierenden Maschinencodes ist dabei wie in Anhang B unter Abbildung B.3 vorgestellt.
Um einen Übersetzungsvorgang inkl. Scheduling durchführen zu können benötigt der Assembler folgende Informationen:

- Rechenzeiten der einzelnen ALU-Operationen
- Verzögerungszeiten der verwendeten Pipelines
- Zuordnungen der Assemblerbefehle zu Binärcode
- Zusammensetzung des Binärcodes (siehe Anhang B Abbildung B.3)
- Optimierungsausdrücke zur Festlegung des Schedulings

Diese Informationen werden in einer Konfigurationsdatei gespeichert. Innerhalb dieser Datei werden dem Assembler die oben genannten Informationen für alle Prozessorkerne zur Verfügung gestellt. Dabei wurde das XML-Dateiformat zur Speicherung der Daten gewählt, da dieses Format von vielen Programmiersprachen mit eigenen Klassen analysiert und damit verwendet werden kann. Damit wird die Anpassbarkeit des ViSARD und die damit einhergehende Konfigurierbarkeit des Assemblers erreicht.
Die Kompatibilität des Assemblercodes der Einkern- und Mehrkern-Version des ViSARD Softcore ist weiterhin gegeben. Das bedeutet, dass sich der Assemblercode für die Einkern-Version nicht von dem Code der Mehrkern-Version unterscheidet. Dadurch kann jeder Assemblercode auf ViSARDs mit unterschiedlichen Konfigurationen in der Kernzahl ausgeführt werden. Ein Resultat daraus ist die Notwendigkeit, dass der Assembler die Zuweisung von Befehlen auf die verschiedenen Prozessorkerne festlegen muss. In Abschnitt 4.4.3 wurden mögliche Verbindungstopologien einer Mehrkernarchitektur diskutiert. Dadurch ergibt sich die zusätzliche Anforderung des Assemblers, entsprechende Konflikte (Deadlocks) von Lese- bzw. Schreibzugriffen auf den gemeinsamen Speicher bereits während des Compilevorgangs zu erkennen und zu beheben. In diesem Zusammenhang muss der Assembler eine Aufteilung der Befehle auf die vorhandenen Kerne vornehmen und sicherstellen, dass zu den jeweiligen Zeitpunkten des Starts der Operationen auf jedem Kern alle notwendigen Argumente verfügbar sind. Dabei können Deaklocksituationen auftreten, die der Assembler entsprechend auflösen muss. Diese Deadlockbehandlung ist Thema des Abschnitts 5.3.3 „Scheduling für Multi-Core ViSARD“.
Aufgrund der Eigenschaft zur Designzeit taktgenau vorhersagbar zu sein, sind im resultierenden Maschinencodes (und damit bereits im Assemblercode) Sprungbefehle verboten. Da für viele Programme allerdings Schleifen benötigt werden und ein reines Ausrollen aller intern verwendeten Schleifen zu einer starken Vergrößerung des Speicherbedarfs des resultierenden Softcore Befehlscodes (*Softcore Instruction Code*, siehe Abbildung 5.21) führen würden, bietet der ViSARD die Möglichkeit einer Hardwareschleife. Mit dieser kann an einem durch den Assembler zur Designzeit festgelegten Punkt im Maschinencode gesprungen werden. Die Hardwareschleife wurde im Abschnitt 5.1 vorgestellt. Um diese nutzen zu können, benötigt der Assembler Informationen darüber, welche Teile des Assemblercodes innerhalb und welche Teile vor oder nach der Schleife liegen. Das wird über die Möglichkeit gelöst, dass dem Assembler beliebig viele Assemblercodedateien für einen Compilevorgang gegeben werden und über den Befehlszusatz „loopable“ als Argument bei den entsprechenden Dateien (*Assembly Code File 3*, siehe Abbildung 5.23). Der Assembler wird während des Übersetzungsvorgangs die einzelnen Dateien intern kombinieren, diese allerdings so in Maschinencode übersetzen, dass alle für eine Schleife spezifizierten Assemblercodedateien im resultierenden Maschinencode in einem separierten Block vorliegen (*Softcore Instruction Part 3*) und Eintritts- sowie Austrittspunkte der Blöcke als Informationen für den Softcore zur Verfügung gestellt werden. Nicht schleifenfähige Assemblercodes (*Assembly Code File 1* und *Assembly Code File 2*) werden sich zur Optimierung der Pipelinenutzung potenziell im resultierenden Maschinencode

überlappen (*Softcore Instruction Code Part 1* und *Softcore Instruction Code Part 2*), wie in Abbildung 5.23 illustriert.

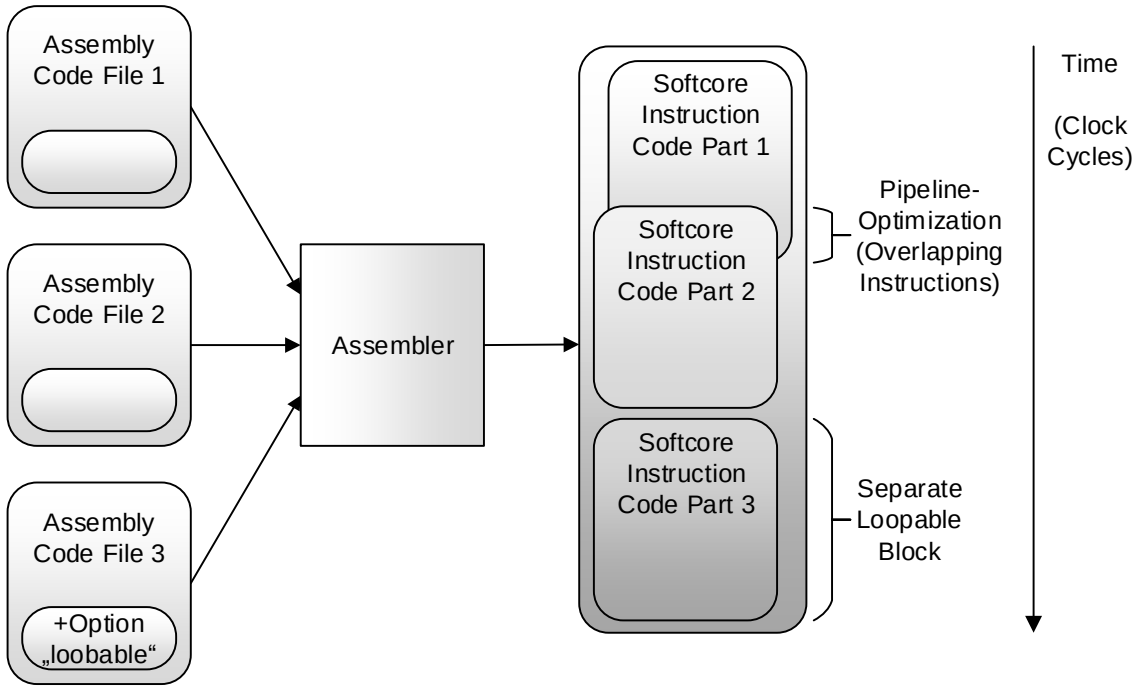



Abbildung 5.23: Kombination mehrerer Eingabe-Assemblercodedateien

Der Assembler stellt sicher, dass alle Operationen, die in Assemblercodedateien vor der Schleife definiert werden, vom Prozessor fertig abgearbeitet und die Ergebnisse im Speicher abgelegt sind, bevor ein weiterer Befehl gestartet wird, der sich innerhalb der Schleife befindet. Gleiches gilt für alle Befehle innerhalb der Schleife. Auch diese werden komplett abgearbeitet und die Ergebnisse abgespeichert, bevor eine weitere sich nicht in der Schleife befindliche Operation gestartet wird. Es befindet sich also zum Zeitpunkt des Starts des ersten Befehls in der Schleife kein Befehl mehr in der Prozessorpipeline. Genauso wird sichergestellt, dass alle Befehle fertig abgearbeitet sind, die vor dem Schleifenblock liegen, bevor die Hardwareschleife im ViSARD gestartet wird. Sollte sich zum Zeitpunkt des Starts der Schleife noch eine Operation in der Befehlspipeline des Prozessors befinden, kann dies beim erneuten Start der Schleife zu unvorhersagbarem Verhalten führen. Über den vorgestellten Mechanismus wird das verhindert.
Diese Hardwareschleife hat eine im Prozessor festgelegte Anzahl an Iterationen, dadurch ist der resultierende Maschinencode nach wie vor voll zur Designzeit analysierbar und es kann eine Überprüfung auf Einhaltung der Echtzeitkriterien erfolgen, bzw. sichergestellt werden.

### 5.3.3 Optimierungsverfahren mit unterschiedlichen Zielfunktionen

In diesem Abschnitt werden die verschiedenen umgesetzten Optimierungsverfahren vorgestellt.

### Speicherverwaltung

Die ursprüngliche Version des Assemblers, die bereits in [KKSF17] veröffentlicht wurde, geht von der Annahme aus, dass einer Variablen eine feste Speicherzelle zugewiesen wird. Eine Folgerung aus dieser Annahme ist der Fakt, dass eine Variable zu einem Zeitpunkt immer nur einen Wert annehmen kann. Innerhalb dieser Arbeit wurde bereits die Idee vorgestellt, Variablen als Objekte und nicht als feste Speicherzellen zu betrachten. Wie in Abschnitt 4.5.1 bereits im Detail erläutert, sind in Multi-Graphen Variablen eigenständige Objekte mit Quell- und Zielfunktionen. Diese Idee aus den Multi-Graphen wird innerhalb des Assemblers weiter verfolgt, auch hier werden Variablen im erweiterten Sinn als eine Klasse betrachtet. Der Vorteil liegt darin, dass es möglich ist, diese Klasse zu instantiieren.
Dadurch, dass eine Variable nicht mehr an eine Speicherzelle gebunden ist, kann diese zu einem bestimmten Zeitpunkt in verschiedenen Werten vorliegen. Das hat eine erhebliche Verkürzung der Ausführungszeiten des resultierenden Maschinencodes zur Folge, wie in Abschnitt 5.3.4 anhand verschiedener Benchmarks nachgewiesen wird. Es bietet aber auch die Möglichkeit, eine Variable zwischen mehreren Prozessorkernen zu teilen, ohne die Notwendigkeit zu schaffen, jede Variable unter allen Kernen permanent synchronisieren zu müssen.

Dabei wird die unter Abschnitt 4.6.2 vorgestellte Methode realisiert und jeder Wert einer Variable kann in einer separierten Instanz der Variablen-Klasse (*Variable-Value-Object*) gespeichert werden.
Hierbei muss sichergestellt werden, dass die Korrektheit des Maschinencodes in jedem Fall gewährleistet ist, indem jede Operation exakt den Wert der jeweiligen Variable verwendet, den diese Operation auch mit dem bisherigen Ansatz verwendet hätte. Dazu speichert jede Instanz einer Variable innerhalb des Wert-Objekts (*Value-Object*), wann diese produziert (geschrieben) und konsumiert (gelesen) wird und auf welcher Speicherzelle diese Instanz der Variable abgespeichert ist. Zusätzlich ist es Teil der Informationen, welcher Kern den jeweiligen Wert produziert hat und ob die Instanz der Variable in den geteilten Speicher geschrieben oder mit einem anderen Kern synchronisiert wurde. Damit ist es möglich jeden Wert einer Variable individuell zu verfolgen. Diese Informationen sind über Methoden manipulierbar, d. h. es gibt Methoden um auf diese Informationen lesend und schreibend zuzugreifen. Jede Instanz einer Variable, also jeder mögliche Variablenwert, wird dabei mindestens solange in der Speicherzelle vorgehalten, bis die jeweilige Instanz das letzte Mal konsumiert wurde. Variablen sind dabei nicht an eine Speicherzelle gebunden, sondern jeder Wert, also jede Instanz, besitzt eine eigene Adresse und damit eine eigene Speicherzelle. Sobald eine Instanz das letzte Mal konsumiert wurde, kann diese durch eine neu produzierte Instanz einer anderen Variable ersetzt werden. So wird eine optimale Speicherauslastung und Speicherausnutzung gewährleistet. Ein zu betrachtender Sonderfall sind bedingte Kopierbefehle. Diese kopieren einen Variablenwert, abhängig von einer Bedingung, in eine andere Variable. Sollte diese Bedingung allerdings nicht erfüllt sein, muss sichergestellt werden, dass der vorherige Variablenwert vorliegt. Die Ausgabe des bedingten Kopierbefehls nutzt immer die selbe Speicherzelle wie der vorherige Variablenwert der potenziell zu überschreibenden Variable. Damit ist sichergestellt, dass in jedem Fall von den folgenden Operationen der korrekte Wert gelesen wird.
Ein weiterer Sonderfall ist die Schleifenfähigkeit von bestimmten Teilen des Assemblercodes. Diese Teile werden dem Assembler als separierte Assemblercodedateien übergeben und müssen ebenfalls bei der Übersetzung gesondert behandelt werden. Da bei diesen Codeteilen davon ausgegangen wird, das sie innerhalb des Softcore mehrere Durchläufe absolvieren werden, muss sichergestellt werden, dass die Variablenkonsistenz zwischen zwei Durchläufen gewährleistet

ist. Dieses Problem wird gelöst, indem der Assembler den letzten Wert einer Variable immer auf die selbe Speicherzelle schreibt, die der erste zu lesende Wert der betreffenden Variable verwendet. Variablen auf die schreibend zugegriffen wird, bevor diese gelesen werden, müssen nicht gesondert betrachtet werden. Dabei wird sichergestellt, dass diese Instanz der Variable, also der Variablenwert der als letzter innerhalb dieses Teilprogramms geschrieben wird, nicht durch eine andere Instanz einer Variable überschrieben wird, auch wenn kein weiterer Konsum des Wertes innerhalb des gesamten Programms erfolgt.

### Schedulingverfahren

Bei der aktuellsten Version des umgesetzten Assemblers gibt es keine fest vorgegebenen Schedulingverfahren mehr. Stattdessen wurde ein „Bestrafungssystem“ umgesetzt, das es ermöglicht innerhalb der Konfigurationsdatei speziell auf ein Problem angepasste Verfahren umzusetzen. Dabei wurde die in Abschnitt 4.6.4 vorgestellten Methoden praktisch realisiert.
Durch dieses System kann sich ein Scheduling auf spezielle Eigenschaften des Assemblercodes anpassen. Das geschieht über die Definition von Bestrafungsausdrücken (*Penalties*), die vom Assembler analysiert und zu einem Schedulingalgorithmus zusammengesetzt werden. Ein Scheduling wird also gebildet, indem die Ausführbarkeit einer Instruktion nachgewiesen wird und alle zu einem Zeitpunkt ausführbaren Operationen nach dem Bestrafungswert aufsteigend gescheduled werden. Das erfolgt, wie die Abbildung 5.24 illustriert.

Zu Beginn werden alle Befehle aus den Assemblercodedateien geladen (*Load Instructions from Assembly Files*). In einem ersten Schritt wird überprüft, ob eine (oder mehrere) Assemblercodedateien mit der Option „loopable“ versehen worden sind. Ist das der Fall, werden alle in diesen Dateien stehenden Befehle als abhängig zu allen vorhergehenden Befehlen gesetzt. Weiterhin werden alle Befehle aus folgenden Dateien als abhängig zu den Befehlen der schleifenfähigen Dateien gesetzt. Da eine Operation (im Assemblercode durch den jeweiligen Befehl repräsentiert) erst ausgeführt werden kann, wenn keine Abhnängigkeiten zu vorhergehenden Befehlen mehr vorhanden sind, wird so sichergestellt, dass alle Befehle komplett abgearbeitet und damit nicht mehr in der Befehlspipeline sind, bevor der erste Befehl der Schleife abgearbeitet wird. So wird ein separierter Block aus Befehlen (Operationen) erstellt. Im Anschluss werden alle Befehle markiert die ausgeführt werden können, weil sie keine Abhängigkeiten besitzen. Von allen markierten Befehlen wird ein Bestrafungswert berechnet, der sich aus dem Bestrafungsausdruck ergibt, den der Nutzer im Vorfeld definiert hat. Dieser Wert wird für jede Kombination Kern-Befehl separiert berechnet. Zuerst wird überprüft ob der entsprechende Kern diesen Befehl überhaupt ausführen kann oder ob der Operator in diesem Kern nicht vorhanden ist. Im Weiteren werden Parameter wie Verfügbarkeit der benötigten Argumente lokal oder nur global, oder freie Speicherplätze auf dem Kern, multipliziert mit dem jeweiligen Wert aus dem Bestrafungsausdruck, zur weiteren Berechnung des Bestrafungswertes verwendet. Sind alle Informationen gesammelt, wird der Bestrafungswert für alle Kerne einzeln berechnet. Sind alle Bestrafungswerte aller Befehle auf allen Kernen berechnet, wird der Befehl mit dem geringsten Bestrafungswert ausgewählt und auf dem am besten geeigneten Kern zur Berechnung gescheduled. Alle nur von diesem Befehl abhängigen Befehle werden nach der Ausführungszeit des aktuell geschedulden Befehls als ausführbar markiert und alle Bestrafungswerte werden erneut berechnet. Dieser Vorgang wird wiederholt, bis alle Befehle im resultierenden Scheduling geschrieben sind.
Die Reihenfolge der Abarbeitung der Operationen kann sich dabei aufgrund der Optimierung

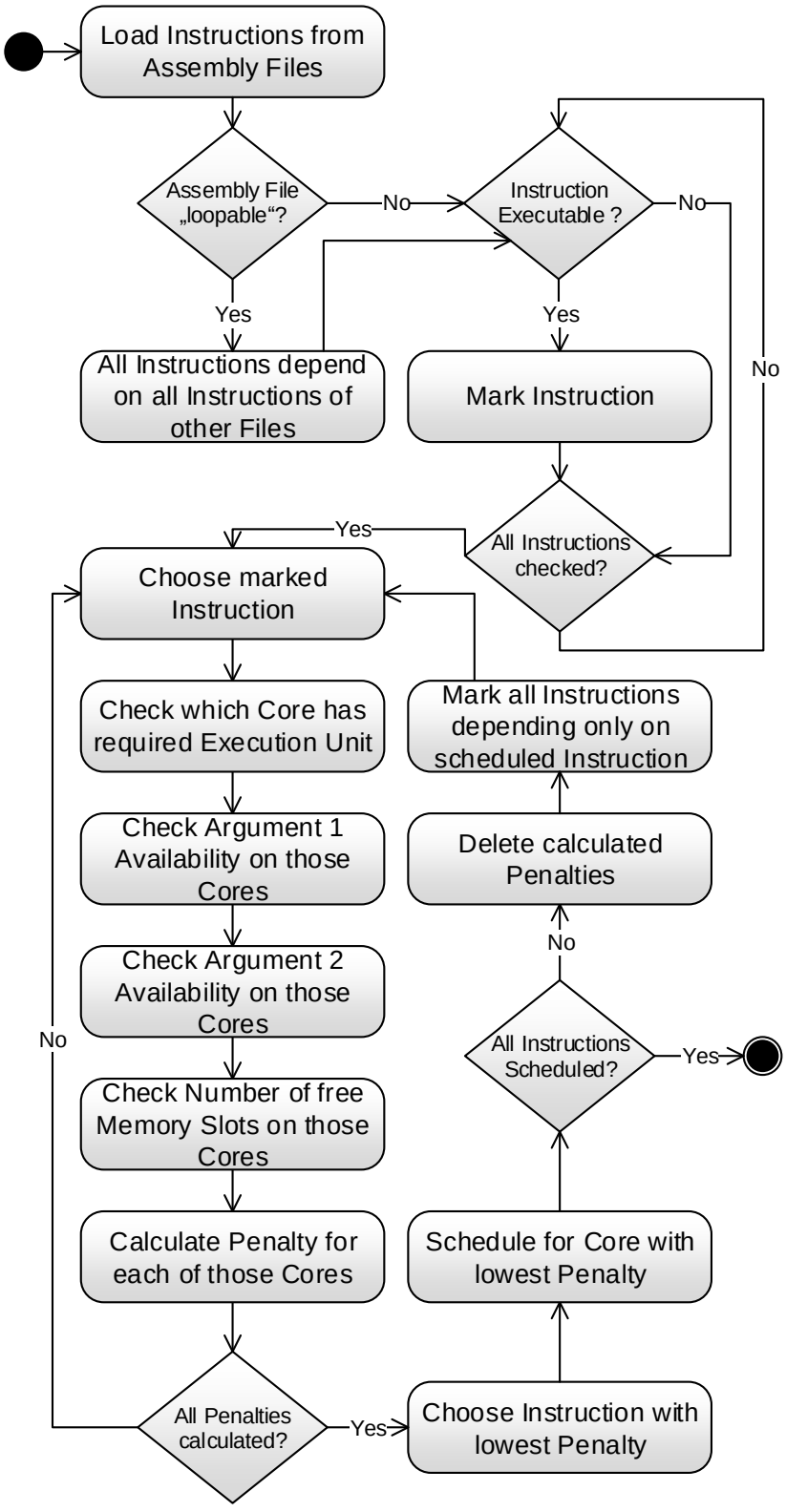



Abbildung 5.24: Funktionsweise des ViSARD Assemblers

in der Pipelinenutzung und Prozessorauslastung von der im Assemblercode definierten Reihenfolge unterscheiden. Sollte die ViSARD Hardwareschleife verwendet werden, so werden die Operationen dabei in Gruppen eingeteilt. Es wird eine Gruppe erstellt für alle Operationen vor den jeweiligen Schleifen, eine Gruppe für jede Hardwareschleife und eine Gruppe für alle Operationen nach der Hardwareschleife, über die zuvor erläuterten Abhängigkeiten. Das Scheduling erfolgt dabei in allen Gruppen getrennt voneinander, sodass sichergestellt ist, dass keine Operation innerhalb oder außerhalb einer Schleife ausgeführt wird, wenn diese im Assemblercode nicht im entsprechenden Teil definiert ist. Dies wird über virtuelle Abhängigkeiten der einzelnen Befehle der jeweiligen Gruppen zu allen Befehlen der anderen Gruppen realisiert.

Dieses Verfahren bietet den Vorteil, dass jedes Scheduling genau auf die Eigenschaften der eingegebenen Assemblercodes angepasst werden kann und die Pipelineauslastung damit verbessert wird. Bei der Verwendung von fest vorgegebenen Schedulingverfahren werden Annahmen über die Problemgröße, Problemklasse und Eigenschaften des zu übersetzenden Assemblercodes getroffen. Diese Annahmen sind statisch und nicht auf das jeweilig spezielle Problem anpassbar. Dadurch sind die resultierenden Schedulings nicht optimal. Aus diesem Grund wurden entsprechend statisch vorgegebene Verfahren nicht verwendet.
Die Tabelle 5.20 zeigt alle Bestrafungsausdrücke, die der Algorithmus verwenden und der Nutzer auswählen kann.

Jeder dieser Ausdrücke kann entweder als Bestrafung, und damit einer Benachteiligung entsprechender Operationen, oder mit negativem Vorzeichen, als Belohnung und damit Bevorzugung entsprechender Operationen verwendet werden. Eine beliebige Kombination aus Bestrafungsausdrücken mit und ohne Vorzeichen, sowie eine Gewichtung jedes einzelnen Ausdrucks ist möglich. Damit der Nutzer nicht gezwungen ist selbst entsprechende Bestrafungsausdrücke zu entwerfen und auf vorhandene Beispiele zurückgreifen kann, wurden einige Beispiele in der Konfigurationsdatei vordefiniert, die verwendet werden können. Diese können auch als Anleitung dienen, eigene Ausdrücke zu definieren. Im Listing 5.1 wird ein Beispiel gezeigt.

| Name | Werte-bereich | Beschreibung |
|---|---|---|
| Instruction.Delay | 0..n | Rechenzeit der Operation in der ALU |
| Operand1. Read-OperationsLeft | 1..n | Anzahl der Operationen, die den ersten Operand dieser Operation als Argument haben |
| Operand2. Read-OperationsLeft | 1..n | Anzahl der Operationen, die den zweiten Operand dieser Operation als Argument haben |
| Operand3. Read-OperationsTotal | 0..n | Anzahl der Operationen, die das Ergebnis dieser Operation als Argument haben |
| Operand1. IsVariable | 0..1 | überprüft, ob das erste Argument eine Variable ist |
| Operand1. IsNoVariable | 0..1 | überprüft, ob das erste Argument keine Variable ist |
| Operand1. Availability.Local | 0..1 | überprüft, ob das erste Argument im lokalen Speicher verfügbar ist |
| Operand1. Availabili-ty.Shared | 0..1 | überprüft, ob das erste Argument im geteilten Speicher verfügbar ist |
| Operand1. Availability. Requestable | 0..1 | überprüft, ob das erste Argument im lokalen Speicher geschrieben werden kann |
| Operand2. IsVariable | 0..1 | überprüft, ob das zweite Argument eine Variable ist |
| Operand2. IsNoVariable | 0..1 | überprüft, ob das zweite Argument keine Variable ist |
| Operand2. Availability.Local | 0..1 | überprüft, ob das zweite Argument im lokalen Speicher verfügbar ist |
| Operand2. Availabili-ty.Shared | 0..1 | überprüft, ob das zweite Argument im geteilten Speicher verfügbar ist |
| Operand2. Availability. Requestable | 0..1 | überprüft, ob das zweite Argument im lokalen Speicher geschrieben werden kann |

Tabelle 5.20: Verwendbare Bestrafungsausdrücke (nach [Wag17])

```
<Penalty>
  <Name> RROmax </Name>
  <Description> Prioritize instructions with results that are
                required often </Description>
  <Expression>  Instruction.Delay-
                100*Operand3.ReadOperationsTotal  </Expression>
</Penalty>
```

Listing 5.1: Beispiel für eine Schedulingstrategie

Folgende Schedulingverfahren sind als Beispiele in der Konfigurationsdatei vorhanden:

- First Come First Served
- Shortest Job First
- Longest Job First
- EmptyMemory
- RROmax

Eine komplette Liste mit diesen vordefinierten Bestrafungsausdrücken ist in Anhang E.1 zu finden.
Soll nun ein neuer Schedulingausdruck definiert werden, geschieht das nach folgenden Schritten: Zuerst muss definiert werden, welche Eigenschaften der resultierende Ausdruck bestrafen soll. Exemplarisch wird hier angenommen, bei einer Mehrkernarchitektur soll die Kommunikation zwischen den einzelnen Prozessorkernen minimiert werden. Dazu müssen Variablen bestraft werden, die nicht in dem lokalen Kern verfügbar sind (bzw. eine Speicherzelle haben), also alle Variablen auf einem anderen Kern sowie im geteilten Speicher. Die Abfrage, ob ein Wert auf einem Kern verfügbar ist, liefert lediglich eine Null oder Eins als Ergebnis, somit sollte dieses mit einer entsprechenden Gewichtung belegt werden. Realisiert wird das über eine Multiplikation des Ergebnisses. Dabei kann die Verfügbarkeit beispielsweise des ersten Operanden mit *Operand1.Availability.Local* auf lokale Verfügbarkeit getestet werden. Analog gilt das für den zweiten Operanden. Die letzte notwendige Information für den Ausdruck ist Ausführungszeit des Befehls mit der längsten Rechenzeit, in diesem Fall exemplarisch eine Division mit 40 Takten. Soll der Algorithmus also in jedem Fall lokale Verfügbarkeit vorziehen, so muss der verwendete Multiplikator größer als 40 sein. Ein resultierender Schedulingausdruck, der immer den kürzesten lokalen Befehl bevorzugt ist in 5.3 nach den diskutierten Regeln zu finden.

$$Instruction.Delay - Operand1.Availability.Local * 41 - Operand2.Availability.Local * 41 \quad (5.3)$$

Diese Bestrafungsausdrücke dienen aktuell vor allem dazu zu ermitteln, wie wichtig eine entsprechende Operation in Relation zu folgenden Operationen ist und ob es etwaige Probleme mit der Verfügbarkeit der notwendigen Instanz der benötigten Variablen als Argumente gibt. Zukünftig wäre es denkbar, dieses System zu erweitern und weitere Bestrafungsausdrücke zu definieren. Denkbar wäre beispielsweise eine Verwendung der Informationen, zu welchen Zeitpunkten wieviele Speicherzellen verfügbar sind, um ggf. Operationen zu bevorzugen, nach denen Variableninstanzen überschrieben werden können um so Speicher für neue von Operationen generierte Variableninstanzen zu schaffen.

#### Scheduling für Multi-Core ViSARD

Ein mögliches Einsatzszenario des ViSARD ist die Verwendung als Multi-Softcore Prozessor. Unter Abbildung 5.24 wurde bereits vorgestellt, dass der Assembler in der Lage ist ein Scheduling für mehrere Prozessorkerne zu erstellen und wie die prinzipielle Funktionsweise dahinter ist. In diesem Abschnitt soll auf diese Mechanismen im Detail eingegangen werden. Bei einem

Scheduling auf mehrere Kerne ist es notwendig, dass der Assembler das gegebene sequentiell geschriebene, Assemblerprogramm automatisch auf die Anzahl an Prozessorkernen aufteilt, welche die jeweilige Konfiguration des Softcores verwendet. Die Betrachtung von Variablen als Klassen und die Werte der Variablen als Instanzen dieser Klasse ist dabei von Vorteil, da dadurch eine Aufteilung der Befehle und Variablen nicht zu einem hohen Overhead an Synchronisationsaufwand führt.
Es kommen dabei die gleichen Schedulingverfahren zum Einsatz, wie auch in der Einkernversion des ViSARD. Dabei wird zusätzlich für jeden Befehl eine Abfrage erstellt, ob alle notwendigen Argumente auf dem jeweiligen Kern verfügbar sind. Ist dies der Fall, kann die Operation gescheduled werden. Sollte bis zu ein Argument nicht auf dem lokalen, aber dafür auf dem gemeinsamen Speicher verfügbar sein und dieser im aktuellen Takt nicht von einem anderen Kern gelesen werden, so kann diese Operation ebenfalls gestartet werden. Fehlen beide Argumente oder ist mindestens ein Eingabewert weder auf dem lokalen noch auf dem gemeinsamen Speicher verfügbar und dieser Kern hat dennoch den geringsten Bestrafungswert, müssen diese Werte angefordert werden. Wie in Anhang B gezeigt, kann das Ergebnis einer Operation gleichzeitig sowohl in den lokalen, als auch den gemeinsamen Speicher geschrieben werden. Entsprechend nicht verfügbare Variablenwerte werden also angefordert, indem die Operation, welche die jeweiligen Ergebnisse erzeugt, diese zusätzlich in den gemeinsamen Speicher schreibt. Da das Scheduling zur Designzeit und nicht zur Laufzeit festgelegt wird, können entsprechende Operationen anderer Kerne so angepasst werden, dass diese die Ergebnisse zusätzlich im gemeinsamen Speicher ablegen, obwohl diese z. B. bereits vor dem aktuellen Befehl gescheduled wurden. Der gemeinsame Speicher verwendet dabei eine Architektur, die es erlaubt in jedem Takt einen Wert zu lesen und einen Wert zu schreiben. Entsprechend ist es nicht möglich eine Operation auf einem Kern zu starten, die zwei Argumente aus dem gemeinsamen Speicher benötigt. Gegebenenfalls würde der Assembler zuerst mittels eines zusätzlich eingefügten „MOV“-Befehls eines der Argumente in den lokalen Speicher kopieren. Weiterhin ergibt sich aus dieser Architektur, dass keine zwei Kerne im gleichen Takt einen Wert in den gemeinsamen Speicher schreiben können. Der Assembler wird also entsprechend alle Operationen aller Kerne so schedulen, dass nie auf zwei Kernen gleichzeitig je ein Ergebnis generiert wird, wenn beide Ergebnisse in den gemeinsamen Speicher geschrieben werden müssen.
Dabei kann es zu Deadlock Situationen kommen, die der Assembler entsprechend auflöst, wie im Folgenden im Detail erläutert wird:
Die Mechanismen zur Deadlock-Erkennung und -Auflösung wurden im Rahmen der Masterarbeit von Herrn Wagner [Wag17] praktisch umgesetzt.
Bei einem Deadlock warten mindestens zwei Befehle auf das Ergebnis des jeweils anderen.

Wie in Abbildung 5.25 zu sehen ist, kann die Addition der Variablen $D$ und $G$ nicht ausgeführt werden, da jeder Prozessorkern (*Core*) jeweils nur einen Operanden im lokalen Speicher verfügbar hat und der jeweils zweite Operand nicht in den gemeinsamen Speicher (*Shared Memory*) geschrieben werden kann, da zu den jeweiligen Takten das Schreiben auf den gemeinsamen Speicher bereits durch den jeweils anderen Kern belegt ist.
Ist auf einem Prozessorkern zu einem bestimmten Takt keine Operation startbar, so wird der Assembler automatisch einen NOP-Befehl einfügen, also wird der Kern im entsprechenden Takt keine Operation starten. Übersteigt die eingefügte Anzahl an NOPs in einem Kern jedoch die maximale Wartezeit die auftreten kann, definiert durch die Rechenzeit der am längsten rechnenden Operation, wird der Assembler einen Deadlock erkennen.
In einem ersten Schritt werden die entsprechend nach der letzten Operation eingefügten NOP-Befehle gelöscht. Weiterhin werden künstliche „MOV“-Befehle eingefügt, um die notwendigen

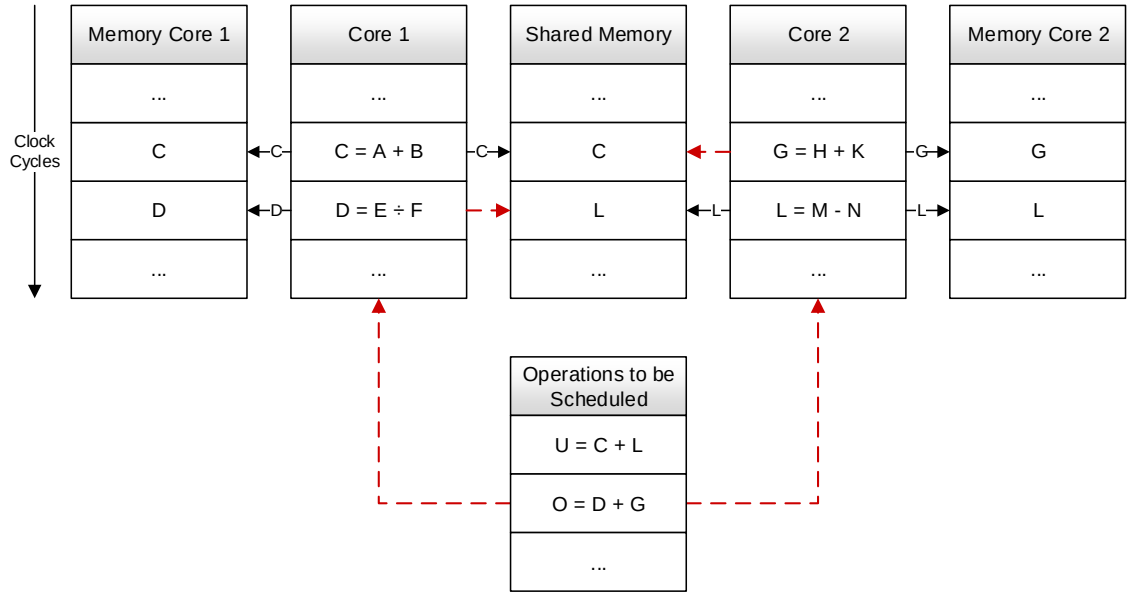



Abbildung 5.25: Beispiel eines Deadlock (angelehnt an [Wag17])

Variablenwerte im gemeinsamen Speicher verfügbar zu machen und so den Deadlock aufzulösen. Ein Spezialfall hierbei liegt darin, wenn die Ausgabe einer Operation bzw. das Ausführen, im Fall des in Abbildung 5.25 gezeigten Beispiels die Addition mit der Ergebnisvariable $O$, auf einem festgelegten Prozessorkern erfolgen muss, da beispielsweise nur dieser Kern die entsprechende Operationseinheit in der ALU besitzt. Sollte auf diesem Kern keine der beiden Variablen lokal vorliegen, dann liegen nach der Deadlock Behandlung zwar beide Argumente im gemeinsamen Speicher vor, aber es kann pro Takt immer nur exakt eine Variable von dort gelesen werden. Aufgelöst wird dieser Konflikt, indem zuerst eine der beiden Variablen auf den lokalen Speicher kopiert wird. Anschließend kann die Operation gestartet werden.

### 5.3.4 Experimentelle Benchmarkergebnisse

In diesem Abschnitt werden die Ergebnisse des Benchmarks des Assemblers vorgestellt. Dabei wurde das in Abschnitt 5.3.1 vorgestellte Benchmark Tool verwendet. Es wurden dabei Assemblercodedateien mit einer Länge von 100 bis 1.000 Assemblerbefehlen je Datei erstellt. Um statistisch verwertbare Ergebnisse zu erhalten, wurde jeder Test 100 mal wiederholt und die Ergebnisse arithmetisch gemittelt. 100 Tests mit gleichem Abhängigkeitsgrad entsprechen also einem Testdurchlauf. Bei jedem Testdurchlauf wurde der durchschnittliche Abhängigkeitsgrad um ca. 10 erhöht. Die komplette Testkonfiguration ist in Anhang E.2 aufgelistet. Es wurden insgesamt 23.700 Assemblercodedateien erzeugt und übersetzt. Im Folgenden wird das Ergebnis aller Benchmarks in einer 3D-Ansicht in Abbildung 5.26 dargestellt.

Zu sehen ist, dass der maximal erzielbare durchschnittliche Abhängigkeitsgrad abhängig von der Anzahl an verwendeten Befehlen im jeweiligen Assemblercode ist. Da die Verteilung der Abhängigkeiten praktisch eine Gleichverteilung ist, wird eine bestimmte Mindestanzahl an verwendeten Befehlen benötigt, um ein entsprechendes Maß an Abhängigkeiten zu erzielen.
Die resultierende Abarbeitungsgeschwindigkeit des Maschinencodes, also die Anzahl an benötigten Takten, steigt dabei etwa linear mit steigendem Abhängigkeitsgrad. Es ist

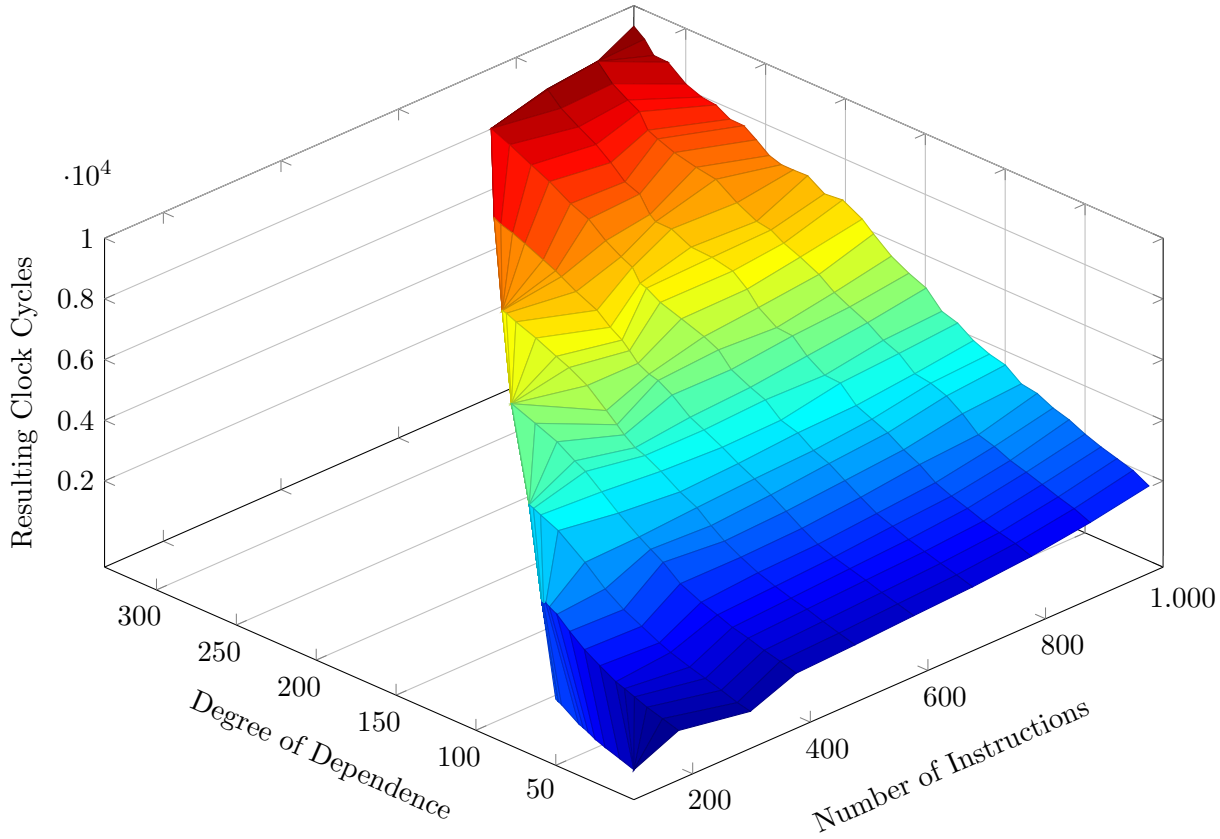



Abbildung 5.26: Ergebnisse des Assembler-Benchmarks

zu erkennen, dass der Anstieg an benötigten Takten bei größeren Assemblercodedateien vergleichsweise geringer ausfällt. Um eine weitere Vergleichsgrundlage zu haben, werden im Folgenden die Durchschnittlichen Takte je Befehl (*Average Clock Cycles per Instruction* (*CPI*)) wie in Gleichung 5.4 aus [PH17] dargestellt, berechnet.

$$CPI = \frac{Resulting\ Clock\ Cycles}{Number\ of\ Instructions} \tag{5.4}$$

Entsprechend werden in Abbildung 5.27 die berechneten CPI-Ergebnisse grafisch dargestellt.

Die Ergebnisse zwischen dem Assemblercode mit 100 Befehlen (*100 Instructions*) und dem Assemblercode mit 1.000 Befehlen (*1.000 Instructions*) werden im direkten Vergleich aus Abbildung 5.28 deutlich.

Während ein Assemblercode mit 100 Befehlen und einem Abhängigkeitsgrad von ca. 50 bereits über 1.300 Takte Ausführungszeit benötigt und damit einen CPI von über 13,4 besitzt, was dem 134-fachen der Anzahl an Befehlen entspricht, benötigt ein Assemblercode mit 1.000 Befehlen bei gleichem Abhängigkeitsgrad nur etwa 2.500 Takte, was lediglich dem 2,5-fachen der Anzahl an Assemblerbefehlen entspricht. Dieses Ergebnis zeigt, dass der Assembler bei sehr komplexen, und damit langen, Assemblercodedateien eine wesentlich effizientere Pipelineauslastung im resultierenden Maschinencode erreicht und ebenfalls die zur Verfügung

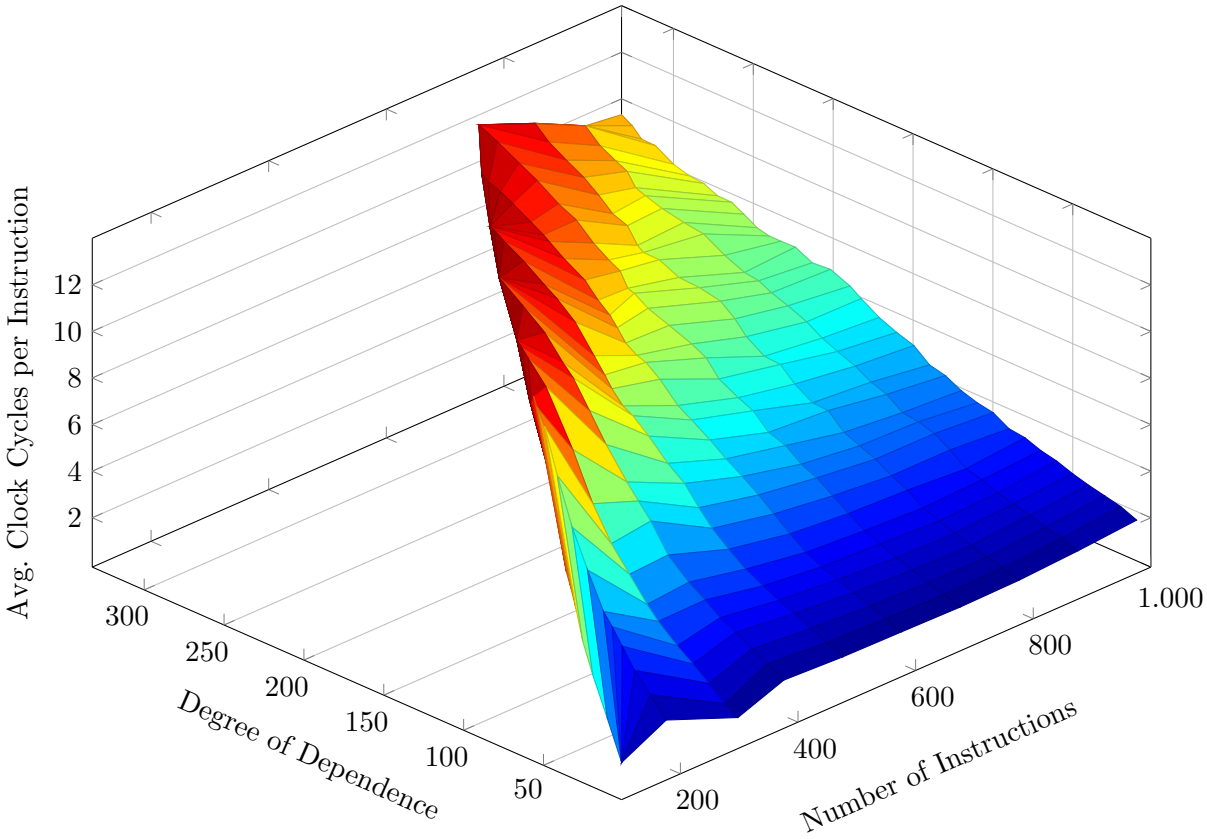



Abbildung 5.27: CPI-Ergebnisse des Assembler-Benchmarks

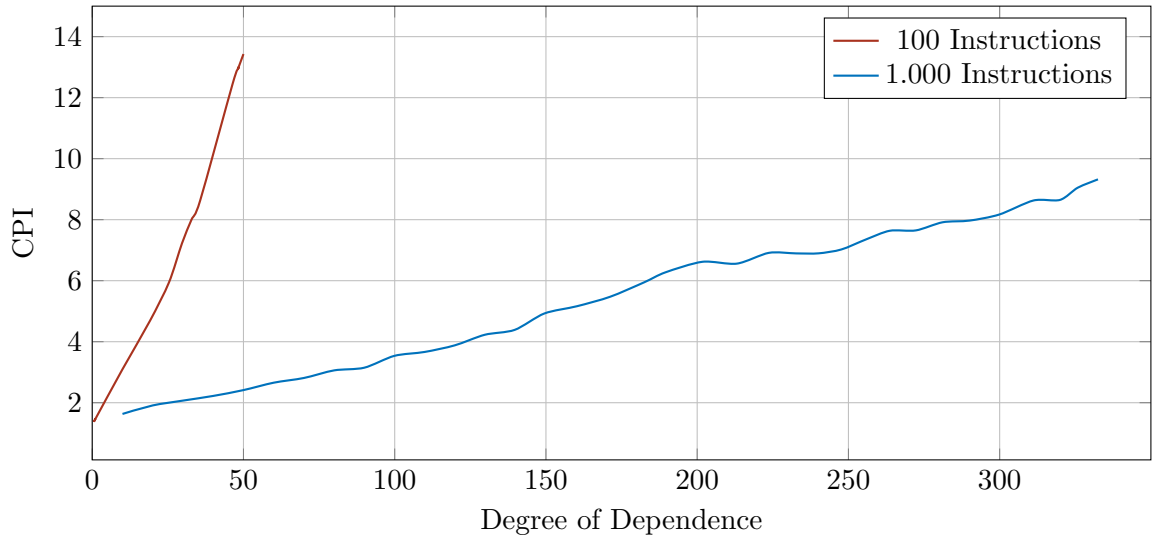



Abbildung 5.28: Vergleich der CPI von 100 und 1.000 Befehlen

stehenden Speicherplätze für Variablen effizienter ausnutzt, was in einem (prozentual zur Länge des eingegebenen Assemblercodes) effizienteren Maschinencode resultiert. Dieses Ergebnis wird bestätigt, wenn man sich die zugehörigen erzielten prozentualen Pipelineauslastungen in der Abbildung 5.29 ansieht. Während die Pipelineauslastung bei 100 Assemblerbefehlen und

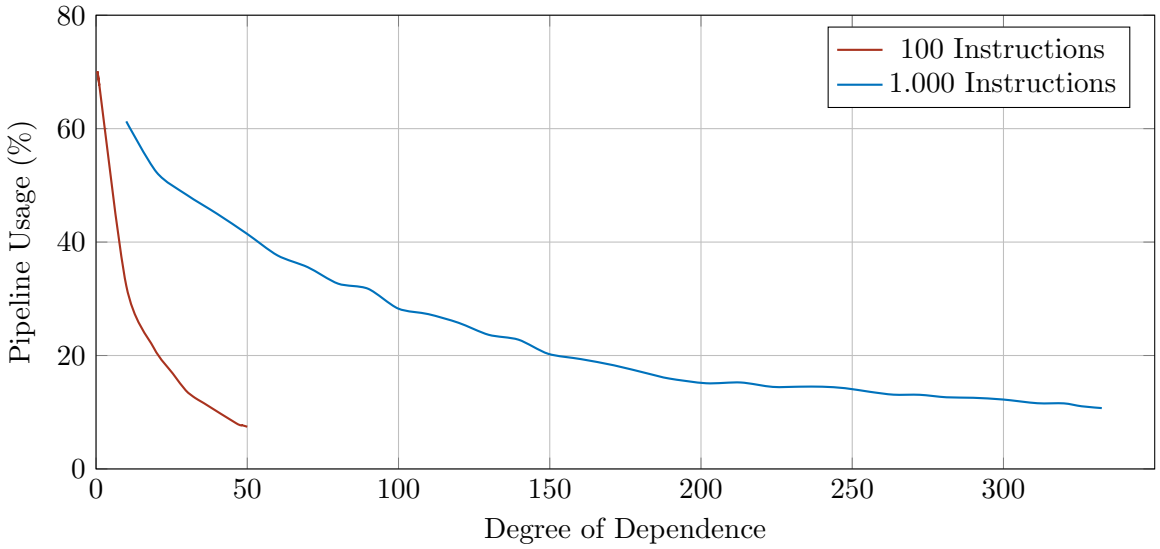
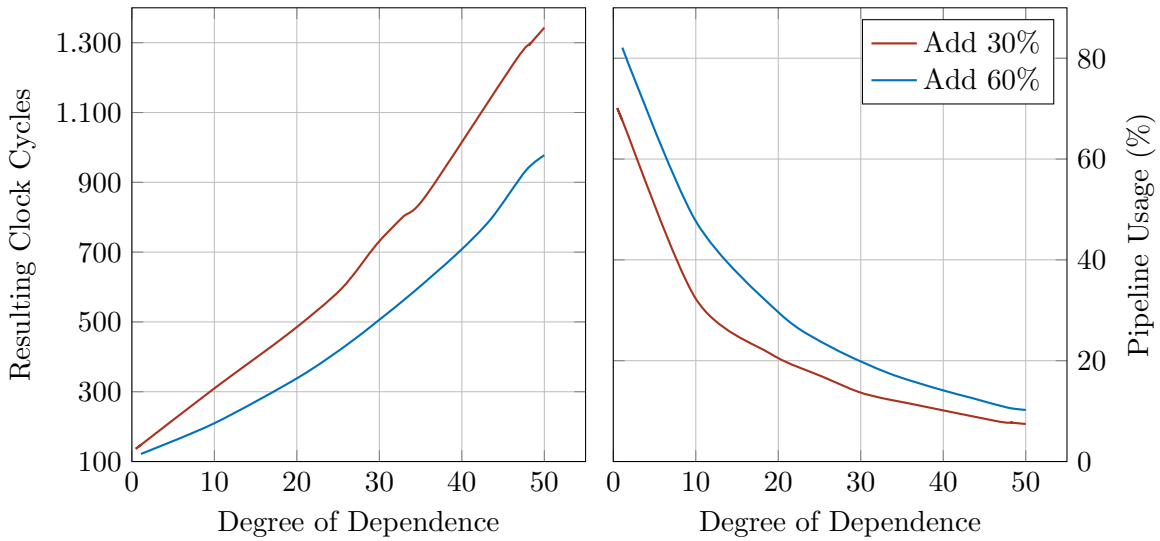



Abbildung 5.29: 1.000 Befehle

einem durchschnittlichen Abhängigkeitsgrad von 50 lediglich noch 7,45 % betragen, liegt die Pipelineauslastung bei 1.000 Befehlen und dem gleichen Abhängigkeitsgrad noch bei 41,42 %. Wie sich die Verteilung der Assemblerbefehle auf den resultierenden Maschinencode und damit auch die resultierende benötigte Anzahl an Takten sowie der erzielten Pipelineauslastung auswirkt wird deutlich, wenn die Auftrittswahrscheinlichkeit der Addition von bisher 30 % auf 60 % verändert wird.




(a) 100 Befehle

(b) 1000 Befehle

Abbildung 5.30: Änderung der Auftrittswahrscheinlichkeit einer Operation

Wie in den Abbildungen 5.30a und 5.30b zu sehen ist, kann das Ergebnis des Assemblers stark von den Eigenschaften des als Eingabe verwendeten Assemblercodes abhängen. Entsprechend wichtig ist die durch die Methodik aus Abschnitt 4.6.4 realisierte Lösung von festen Schedulin-

galgorithmen und die Verwendung von individuell anpassbaren Bestrafungsausdrücken.
Die Experimente zeigen, dass der Assembler in der Lage ist, beliebige Assemblercodes mit theoretisch maximalem Abhängigkeitsgrad zu schedulen und dass das Ergebnisscheduling mit steigendem Abhängigkeitsgrad, lediglich linear mehr Zeit benötigt. Weiterhin haben die Experimente ergeben, dass das Ergebnis stark von den jeweiligen Eigenschaften des eingegebenen Assemblercodes abhängt. Dieses Ergebnis bekräftigt die Notwendigkeit, ein Scheduling dynamisch an die jeweiligen Eigenschaften des aktuell gegebenen Assemblercodes anpassen zu können, um immer ein möglichst optimales Ergebnis zu erzielen. Die durch die vorgestellte Methodik der Bestrafungsausdrücke, aus Abschnitt 4.6.4, realisierten Schedulingverfahren in Kombination mit der vorgestellten Speicherverwaltung, aus Abschnitt 4.6.2, erfüllen diesen Bedarf.
In einem weiteren Test wurde der in Abschnitt 5.2.3 vorgestellte Algorithmus zur Weißlichtinterferometriedatenauswertung (V2 MACG Opt, siehe Tabelle 5.19) übersetzt und mit verschiedenen Einstellungen optimiert. Dieser Assemblercode hat insgesamt 1.070 Befehle, entspricht also von der Anzahl an Befehlen etwa dem längsten im Rahmen des Benchmarks getesteten Assemblercode. Da dieser Algorithmus ein reales Problem löst und nicht lediglich von einem Tool als Benchmark autogeneriert und automatisiert übersetzt wurde, kann das Ergebnis auch sinnvoll mit den Ergebnissen anderer Assembler verglichen werden. Aufgrund der ähnlichen Aufgabenklasse wurde zu diesem Zweck der in [DPZ+13] vorgestellte Assembler, zum zugehörigen Softcore Prozessor „LiSARD“, als Vergleich ausgewählt. Die Ergebnisse sind in Tabelle 5.21 zu finden. Die dort angegebene prozentuale Speicherauslastung ist in Relation zum verfügbaren Speicher von 255 Speicherzellen zu sehen. 255 belegte Speicherzellen entsprechen also einer Speicherauslastung von 100 %.

| | Ausführungs-zeit (%) | Speicher-auslastung (%) | Prozessor-auslastung (%) | Pipeline-auslastung (%) |
|---|---|---|---|---|
| Ohne Optimierung | 100,00 | 22,75 | 5,74 | 6,10 |
| LiSARD Assembler aus [DPZ+13] | 24,18 | 22,75 | 23,74 | 24,55 |
| Pipeline Optimierung | 32,82 | 22,75 | 17,49 | 18,60 |
| Pipeline & Speicher Optimierung | 8,91 | 60,39 | 64,43 | 66,20 |

Tabelle 5.21: Ergebnisse des WLI-Algorithmus

Der Test zeigt deutlich den Unterschied der Ergebnisse, zwischen einer nicht optimierten Version, einer Pipeline-optimierten Version sowie einer Version bei der Pipelineoptimierungen und das verbesserte Speichermanagement zum Einsatz gekommen ist. Die Kombination aus bestrafungsbasierten und damit anpassbaren Schedulingausdrücken von Abschnitt 4.6.4 und der in Abschnitt 4.6.2 vorgestellten Methodik zur Speicherverwaltung erzielt ein stark verbessertes Gesamtergebnis.
Es konnte weiterhin gezeigt werden, dass das Ergebnis des Assemblers mit einer Prozessorauslastung von 64,63 %, im Vergleich zu einem weiteren Assembler, dem Assembler des LiSARDs, mit einer erzielten Prozessorauslastung von 23,74 % besser abschneidet. Der bei diesem Test generierte Maschinencode sowie weitere (zufällig ausgewählte) im Rahmen des Benchmarking erstellte Maschinencodes wurden mittels des Prozessors auf korrekte Funktionsweise überprüft. Alle getesteten Maschinencodes haben wie erwartet funktioniert.

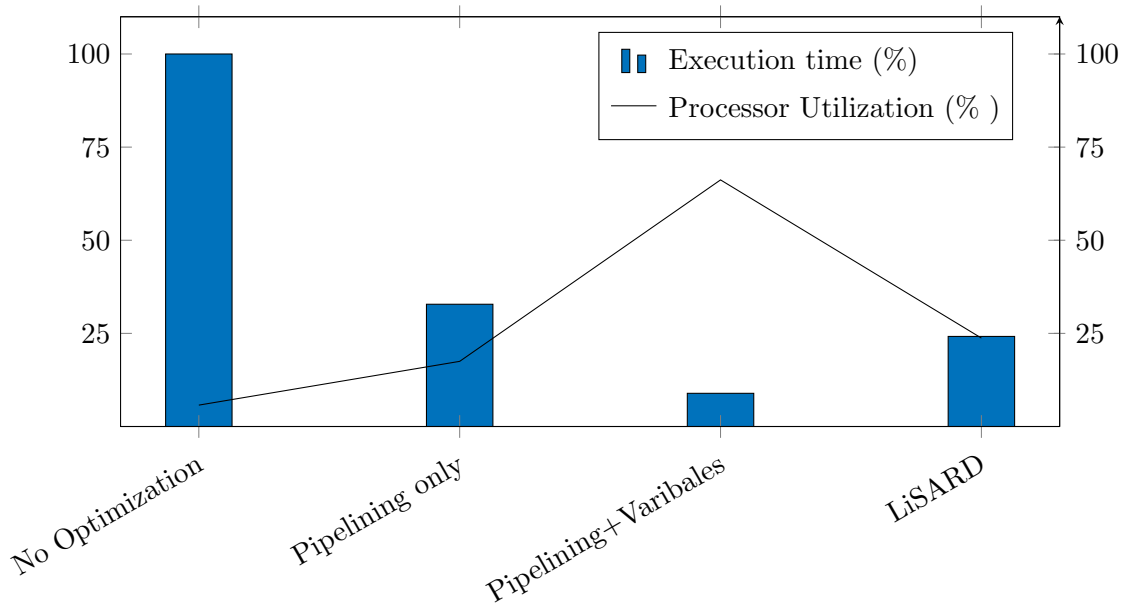



Abbildung 5.31: Vergleich verschiedener Optimierungen des WLI des ViSARD mit dem LiSARD

### 5.3.5 Fazit

Der ViSARD Assembler bietet die Möglichkeit, effizient und automatisiert jeden für den ViSARD geeigneten Assemblercode in Maschinencode zu übersetzen. Während der Übersetzung können optimierende Schedulingverfahren zum Einsatz kommen, die auf spezielle Eigenschaften des jeweiligen Assemblercodes hin angepasst werden können. Mit Hilfe eines Benchmark Tools ist es möglich, die Effektivität neu definierter Schedulingausdrücke zu testen.
Dank der Realisierung, bei der Variablen als instanziierbare Objekte betrachtet werden, ist es möglich zu jedem Zeitpunkt verschiedene Wertigkeiten einer Variable in unterschiedlichen Speicherzellen vorliegen zu haben. Mit diesem Mechanismus können Abhängigkeiten, wie beispielsweise Write-after-Read Abhängigkeiten aufgelöst und der resultierende Maschinencode effizienter umgesetzt werden. Diese Steigerung der Effizienz ist durch eine erhöhte Pipelinenutzung sowie eine daraus resultierende erhöhte Prozessorauslastung messbar. Der Assembler stellt dabei alle notwendigen Eigenschaften des resultierenden Maschinencodes, wie beispielsweise die taktgenaue Vorhersagbarkeit, sicher.
Im Abschnitt 5.3.4 wurden diese umgesetzten Mechanismen quantitativ mittels des vorgestellten Benchmark Tools untersucht. Dabei wurde die Ergebnisse im Speziellen auf das Verhältnis von Abhängigkeitsgrad und resultierender benötigter Rechenzeit hin untersucht. Es wird weiterhin gezeigt, welchen Einfluss die Eigenschaften, also beispielsweise prozentuales auftreten bestimmter Befehle oder Abhängigkeitsgrad der Befehle, des zu übersetzenden Assemblercodes auf die Qualität des Ergebnisses haben. Im Anschluss daran wurde mit einem Algorithmus zur Weißlichtinterferometriedatenauswertung ein reales Problem übersetzt und mit einem anderen Assembler verglichen.
Die Ergebnisse zeigen, dass die umgesetzten Methoden korrekt und effizient funktionieren und dass der resultierende Maschinencode alle notwendigen Eigenschaften der Aufgabendomäne erfüllt. Weiterhin wurde gezeigt, dass der Assembler die notwendigen Eigenschaften hat, die in der vorgestellten Methodologie benötigt werden.

# 6 Zusammenfassung und Ausblick

## 6.1 Zusammenfassung

Die steigende Komplexität von eingebetteten Systemen und das damit verbundene wirtschaftliche und wissenschaftliche Entwicklungspotenzial stellt einen wichtigen Gegenstand aktueller Forschungen dar. In der vorliegenden Arbeit wurde deshalb eine komplexe Realisierungsmethodik für applikationsspezifische Softcore FPGA-Lösungen in Abhängigkeit von algorithmischen Anforderungen im Einsatzgebiet eingebetteter Systeme entwickelt. Die erarbeiteten Methoden und Techniken zur Software-, Hardware- und Hardware-Software Co-Design Entwicklung mit Softcore Prozessoren ermöglichen immer komplexere Systeme zeiteffizient unter Einsatz von modellbasierten Entwicklungsmethoden zu realisieren. Die in der Arbeit entwickelten Methoden erlauben die hierarchische Beschreibung und Validierung sowie die Optimierung von Funktionsverteilungen in eingebetteten Systemen mit Softcore Prozessoren, durch das erweiterte V-Modell aus Abbildung 4.2 sowie das neu erstellte *Phi*-Vorgehensmodell aus Abbildung 4.1. Aufgrund der steigenden Komplexität gewinnen Mechanismen zur frühzeitigen Validierung und der damit einhergehenden Fehlererkennung sowie Fehlervermeidung, immer mehr an Bedeutung. Weiterhin ist es notwendig, die Wiederverwendung bereits vorhandener und getesteter Logik zu maximieren, um eine Minimierung von Kosten und Entwicklungszeit zu erreichen. Die Kosten werden vor allem durch die Rekonfigurierbarkeit der FPGAs und der damit wegfallenden Herstellungskosten der Hardware erreicht und die Entwicklungszeit wird über die stark verkürzten Entwicklungszyklen bei der Verwendung von FPGAs erreicht. Auch die durch die in dieser Arbeit vorgestellten Modelle und die dort vorgestellten Evaluationszyklen wird eine Verkürzung der Entwicklungszeit erreicht. Die in der Arbeit entwickelten Modelle und Methoden zielen weiterhin darauf ab, die steigende Komplexität besser zu berücksichtigen, indem Mechanismen zur Abgrenzung einzelner kleinerer Arbeitsschritte, sowie deren Validierung umgesetzt wurden. Dadurch ist eine frühzeitige Fehlererkennung sichergestellt.
Die zunehmende Leistungsfähigkeit speziell von FPGAs und FPGA-basierten Plattformen eröffnet immer weitere Einsatzdomänen, in denen bisher deutlich größere Systeme entsprechende Aufgaben übernommen haben. Aus diesem Grund benötigt es neue, in der Arbeit umgesetzte, an die Komplexität angepasste Realisierungsmethodiken.

Im Rahmen dieser Arbeit wurden zunächst in Kapitel 2 für das Verständnis der Arbeit notwendige Grundlagen dargestellt. Es wurde beschrieben, was ein eingebettetes System ist und mit welchen allgemeinen Standardmodellen diese Systeme entworfen werden können. In Abschnitt 2.3 wurde spezieller auf den komponentenorientierten Entwurf, sowie in den weiteren Abschnitten auf den Top-Down, sowie den Bottom-Up Systementwurf eingegangen, da diese Entwurfsverfahren eine wichtige Grundlage für die entwickelte Realisierungsmethodik bei der Entwicklung mit Softcore Prozessoren darstellen. Im folgenden Abschnitt 2.4 wurde der Aufbau und die Funktionsweise von Field Programmable Gate Arrays vorgestellt. Im Speziellen wurde

dabei in Unterabschnitt 2.4.2 auf die partiell rekonfigurierbaren FPGA-Plattformen und die Funktionsweise sowie die Einschränkungen der partiellen Rekonfiguration eingegangen.
Der Abschnitt 2.5 beschäftigt sich mit den Grundlagen von Übersetzungswerkzeugen, wie beispielsweise Assembler und Compiler. Die im Rahmen der Arbeit entwickelten Teile der Toolchain realisieren in wichtigen Punkten entsprechende Übersetzungswerkzeuge. Notwendige Grundlagen zum Verständnis dieser Umsetzungen wurden an dieser Stelle dargestellt. Das Kapitel 2 schließt mit einer Vorstellung von ausgewählten Problemen der echtzeitkritischen Daten- und Bildverarbeitung. Diese Beispiele sind repräsentativ für die in dieser Arbeit adressierten Aufgabendomänen.
Das Kapitel 3 befasst sich mit mit Entwurfsmethoden von Softcore Prozessoren. In diesem Rahmen wurde in Abschnitt 3.1 zuerst ein allgemeiner Gesamtentwurf vorgestellt. Im folgenden Abschnitt 3.2 wurde spezieller auf den Entwurf einzelner IPs innerhalb des Softcore Prozessors eingegangen. Es wurden einzelne Entwurfsschritte beim Entwurf von Softcore basierter Logik in Abschnitt 3.3, sowie ein allgemeines Hardware-Software Co-Design Entwurfsverfahren in Abschnitt 3.4 präsentiert. Am Ende dieses Kapitels ist ein Fazit zur Verwendung von Softcore Prozessoren in der Aufgabendomäne der echtzeitkritischen Daten- und Bildverarbeitung zu finden.

Bei den in dieser Arbeit adressierten Aufgabengebieten ist es notwendig, verbindliche Zusagen über die korrekte und zeitgenaue Abarbeitung von Algorithmen treffen zu können, da das Verletzen von Deadlines schwerwiegende Auswirkungen haben kann.
Die Kombination aus den Anforderungen einer kosten- und zeiteffizienten Entwicklung und dem Nachweis des verbindlichen Einhaltens echtzeitkritischer Schranken scheinen entsprechend gegenläufig. Das Kapitel 4 beschäftigt sich daher mit eigenen Entwicklungsmethoden und -modellen, um diese Anforderungen zu kombinieren. In diesem Zusammenhang wurde in Abschnitt 4.1 zunächst das entwickelte *Phi*-Modell zur Softcoreentwicklung entwickelt. Dieses wurde in Abschnitt 4.2 in ein angepasstes V-Modell eingegliedert. Dieses V-Modell wurde speziell auf die Entwicklung von eingebetteten Systemen mit Softcore Prozessoren angepasst.
Der folgende Abschnitt 4.3 befasst sich mit der aus den Anforderungen der zeit- und kosteneffizenten Entwicklung, sowie der Handhabung immer komplexerer Systeme, abgeleiteten Entwicklungskette (Toolchain). Diese minimiert den Einsatz von Zeit und Geld bei einer Entwicklung und maximiert entsprechend die Möglichkeit der Wiederverwendbarkeit. Ausgelegt ist die Toolchain dabei auf sehr komplexe Systeme.
Die folgenden Abschnitte 4.5 und 4.6 befassen sich mit den entwickelten Übersetzungswerkzeugen. Abschnitt 4.5 geht dabei speziell auf die methodische Assemblercodegenerierung mit Datenflussgraphen ein. Hierbei kommen spezielle Multi-Graphen zum Einsatz. Innerhalb dieses Abschnitts wurde sowohl die Darstellung, als auch die Optimierung entsprechend erstellter Datenflussgraphen diskutiert. Ein weiterer wichtiger Punkt neben der Optimierung des Datenflussgraphen ist die Sequentialisierung des Graphen in Assemblercode. Zur Lösung wurden verschiedene Sequentialisierungsstrategien umgesetzt sowie eine Logik zur Minimierung des durch den hohen Abstraktionsgrad entstehenden Speicheroverheads des resultierenden Assemblercodes. Abschnitt 4.6 befasst sich mit dem Assembler, der die erstellten Assemblercodes in für den Softcore Prozessor lesbaren Maschinencode übersetzt. Innerhalb dieses Abschnitts wurden notwendige Voraussetzungen sowie die auftretenden Variablenabhängigkeiten vorgestellt. Die optimale Speicherverwendung, also das Aufteilen von Variablen auf den physisch vorhandenen Speicher, ist ein Kernpunkt dieses Abschnitts. Hierbei wurde ein Verfahren umgesetzt, bei dem Variablen als instantiierbare Objekte angesehen und entsprechend behandelt werden. Der zweite Kernpunkt dieses

Abschnitts sind verschiedene Schedulingalgorithmen, die eine schnellstmögliche Abarbeitung des Assemblercodes im Softcore Prozessor ermöglichen. Hierbei wurden verschiedene Verfahren zur Maximierung der Pipelineauslastung und damit schnellstmöglichen Abarbeitung, vorgestellt. Die sogenannten penaltybasierten Schedulingausdrücke, die in diesem Abschnitt entwickelt wurden, ermöglichen eine Anpassung des erzeugten Schedulings an die speziellen Eigenschaften jedes vorliegenden Assemblercodes. Die Umsetzung löst sich dabei von fest vorgegebenen Schedulingalgorithmen und realisiert einen sich den speziellen Eigenschaften des Assemblercodes anpassbaren Assembler.
Der Abschnitt 4.4 beschäftigt sich mit dem Konzept eines echtzeitfähigen Softcore Prozessors. Dabei wurde im ersten Unterabschnitt ein allgemeiner Aufbau dargestellt und erläutert. Die folgenden Unterabschnitte befassten sich mit der Erweiterung dieses Grundkonzepts in verschiedenen Richtungen: Erweiterungen zur Minimierung des Energieverbrauchs, Konzepte von Mehrkernarchitekturen sowie ein Konzept zur Minimierung des Ressourcenverbrauchs wurden diskutiert. Das Konzept zur Minimierung des Ressourcenverbrauchs benutzt dabei die bereits vorgestellte partielle Rekonfiguration. Hier wurden sowohl die Voraussetzungen, aber auch die spezifischen Umsetzungen sowie verschiedene Rekonfigurationsgranularitäten vorgestellt.

Das Kapitel 5 stellt alle praktischen Ergebnisse der vorliegenden Arbeit vor. Innerhalb dieses Kapitels wurden alle Module der in Kapitel 4 entwickelten Toolchain umgesetzt und ausgewertet. Der realisierte Softcore Prozessor, der ViSARD, wird in Abschnitt 5.1 vorgestellt. Dieser kann als Einkern- oder Mehrkernarchitektur mit einer Operatorgenauigkeit von aktuell 32-Bit oder 64-Bit betrieben werden. Ein wichtiger Teil dieses Abschnitts ist die partielle Rekonfiguration des ViSARD. Hierfür wurden Experimente in verschiedenen Kombinationen und mit verschiedenen Rekonfigurationsgranularitäten durchgeführt und ausgewertet. Anhand dieser Experimente erfolgt der Nachweis der korrekten Funktionsweise, sowie der genauen zeitlichen Planbarkeit der Rekonfiguration.
Alle aus Abschnitt 4.5 vorgestellten Verfahren zur Datenflussgraphoptimierung und die dort vorgestellten Algorithmen zur Sequentialisierung und Variablenminimierung wurden in Abschnitt 5.2 im Rahmen des Modelbased Assembly Code Generator (MACG) umgesetzt. Der gesamte Funktionsumfang wurde in diesem Abschnitt dargestellt. Insbesondere wurden dabei in Unterabschnitt 5.2.3 die experimentellen Ergebnissen des MACG dargestellt. Hier wurden verschiedene, bereits in anderen Softcore Prozessoren verwendete, Algorithmen umgesetzt und mit den jeweiligen originalen Assemblerprogrammen qualitativ verglichen. Als Vergleichsgrundlage wurden sowohl der benötigte Speicher in Form von Variablen, als auch die Anzahl an resultierenden Codezeilen im Assemblercode und die damit erzielte Softcore-Rechenzeit verwendet.
In Abschnitt 5.3 wurde der umgesetzte Assembler, also das Programm zur Übersetzung des Assemblercodes in Maschinencode, vorgestellt. Der Assembler wurde mit Hilfe eines speziell entwickelten Benchmark Tools sowohl qualitativ, als auch quantitativ getestet. Zusätzlich wurden die Ergebnisse des Assemblers von speziell ausgewählten Assemblercodes mit einem weiteren, vergleichbaren, Assembler eines Softcores der selben Aufgabendomäne verglichen.
Mit den Experimenten aus den Abschnitten 5.2, 5.3 und 5.1 wurde die gesamte Verarbeitungskette der Toolchain, jeweils Abschnitt für Abschnitt getestet. Abschnitt 5.2 beschäftigt sich mit der modellbasierten Assemblercodegenerierung des MACG. Der so erzeugte Assemblercode wird über den in Abschnitt 5.3 vorgestellten und getesteten Assembler in für den Softcore lesbaren Maschinencode übersetzt. Dabei werden für die Softcore-Bibliothek zur Anpassung notwendige Informationen gesammelt. Mit Hilfe dieser Informationen und den

Anforderungen aus der Aufgabenstellung kann aus der Softcore-Bibliothek ein angepasster Softcore erstellt werden. Dieser angepasste Softcore wurde in Abschnitt 5.1 experimentell untersucht. Entsprechend wurden alle notwendigen Schritte der vorgestellten Toolchain praktisch umgesetzt und experimentell untersucht.

Zusammenfassend kann gesagt werden, dass die erarbeitete Realisierungsmethodik von applikationsspezifischen Softcore Prozessoren eine effiziente Entwicklung sowohl in Industrie- aber auch wissenschaftlichen Projekten der vorgestellten Aufgabendomänen ermöglicht. Dabei sind vor allem die Bereiche der echtzeitkritischen Datenverarbeitung und Bildverarbeitung adressiert. Für jede Abstraktionsebene innerhalb dieser Projekte wurden Modelle vorgestellt, die jeweils auf der Abstraktionsebene alle notwendigen Anforderungen sinnvoll umsetzen. Dabei entsteht eine Toolchain, mit dessen Hilfe eine zeit- und kosteneffiziente Entwicklung ermöglicht wird. Über verschiedene Mechanismen wird dabei auch die Wiederverwendbarkeit maximiert. Durch die gezielte Festlegung einzuhaltender Kriterien wird dabei in jedem Schritt der Kette eine bestmögliche Kombination von zeit- und kosteneffizienter Entwicklung mit der Sicherstellung der Einhaltung harter Echtzeiteigenschaften erreicht.
Der praktische Nachweis der Funktionalität der Modelle wird über die Umsetzung der vorgestellten Toolchain erbracht. In diesem Zusammenhang wurden ebenfalls weitere für den Nachweis notwendige Mechanismen und Programme, wie ein Benchmark Tool, umgesetzt. Mit Hilfe dieser Realisierungen wurden Experimente durchgeführt, um die jeweiligen Verarbeitungsschritte qualitativ einordnen zu können. Dabei wurden für jeden Schritt der Toolchain separierte Experimente durchgeführt, um die Ergebnisse des jeweiligen Schritts qualitativ einordnen zu können. Speziell bei den Experimenten zur partiellen Rekonfiguration des Softcore Prozessors konnte ein Nachweis über zeitliche Anforderungen erbracht werden, mit denen das dynamische Austauschen von Teilen des Softcore oder sogar ganzen Softcore Prozessoren zur Laufzeit ermöglicht wird.
Der Kernpunkt der praktischen Umsetzung, der Softcore Prozessor, bietet dabei verschiedenste Anpassungsmöglichkeiten, die ihn zu einem in der Aufgabendomäne praktisch umfassend einsetzbaren und trotzdem spezialisierten Tool machen.

## 6.2 Ausblick

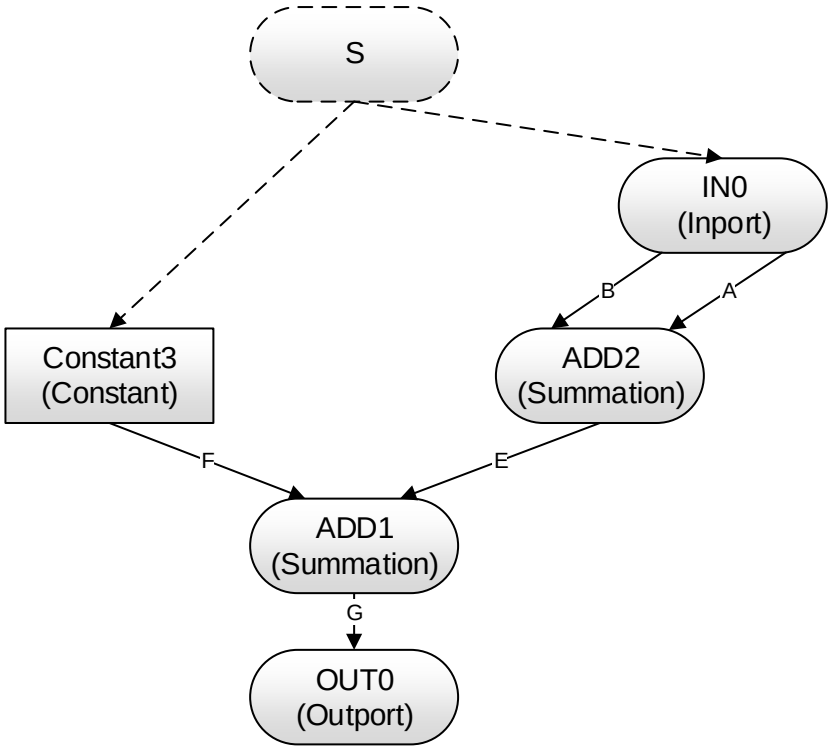
Ausgehend von den für die Dissertation erarbeiteten und in ihr dargestellten Ergebnisse lassen sich weiterführende Arbeiten einordnen.
Die Toolchain, wie sie zum Zeitpunkt dieser Arbeit praktisch umgesetzt ist, bietet einem Entwickler die Möglichkeit eine zu einem erheblichen Teil automatisierte Projektentwicklung und Problemlösung durchzuführen. Da der Prozessor weitestgehend Plattform unabhängig ist, kann dieser auf in Zukunft auf neue FPGA-Plattformen und Familien portiert werden. Eine sinnvolle Erweiterung in diesem Zusammenhang ist die bisher nicht verfügbare ebenfalls möglichst plattformunabhängige automatisierte Realisierungsmöglichkeit von partiellen Rekonfigurationen des Softcores. Diese muss zum Zeitpunkt dieser Arbeit für jeden FPGA und jeden Algorithmus manuell umgesetzt und überprüft werden. Denkbar wäre ein Tool, dass einen gegebenen Maschinencode automatisiert analysiert, etwaige partielle Rekonfigurationsmöglichkeiten berechnet und diese gegebenenfalls automatisiert umsetzt.

Die Modelle und Mechanismen, um einen gegebenen Datenflussgraphen automatisiert zu optimieren, sollten erweitert und verfeinert werden. Vorgeschlagen wird an dieser Stelle die Entwicklung komplexer Methoden, um einen gegebenen Datenflussgraphen mit der Zielfunktion der Minimierung der resultierenden Rechenzeit des erzeugten Assemblercodes zu manipulieren. Hierzu muss analysiert werden, welche Konstrukte zu einer unnötige Verlängerung der Laufzeit führen und diese müssen funktionsäquivalent verändert werden.

Innerhalb des Assemblers ist eine Deadlockerkennung umgesetzt, die zum Einsatz kommt, wenn Programme für eine Multi-Core Anwendung übersetzt werden. Diese Deadlockerkennung kann erweitert und zu einer Deadlockvermeidung ausgebaut werden. Der Nachteil in der aktuellen Realisierung liegt darin, dass sobald ein Deadlock auftritt, dieser zwar behoben wird, er aber zu einer Verringerung in der Pipelineauslastung jedes Kerns führt. Bei einer frühzeitigen Deadlockvermeidung könnte daher die Pipelineauslastung erhöht werden. Es müsste also eine Methode zur Erkennung von Deadlocks entwickelt werden, die das Scheduling der Befehle im Vorfeld des Deadlocks bereits so anpasst, dass die Pipelineauslastung weiterhin sehr hoch ist, ein entsprechender Deadlock aber vermieden wird.
Im Softcore Prozessor selbst könnten weitere Konzepte zur Durchsatzsteigerung, wie beispielsweise ein Bypassing-Konzept, umgesetzt werden, was die Pipelineauslastung weiter steigern würde.

Es sollte die Möglichkeit untersucht werden, die partielle Rekonfiguration möglichst automatisierbar in entsprechende Projekte einzubinden. Hierzu sollten die Modelle und Vorgehensweisen verfeinert werden, um sich auf die speziellen Eigenschaften und daraus ableitbaren notwendigen Schritte bei der Arbeit mit partieller Rekonfiguration besser anpassen und darauf eingehen zu können. In den aktuellen Modellen und Vorgehensweisen ist die partielle Rekonfiguration zwar bedacht, allerdings ist aktuell keiner der notwendigen Schritte automatisiert.

# A Beispiel zum Datenflussgraph




(a) Graph zur Sequenzialisierung

| Step | List | Scheduling Order |
|---|---|---|
| 1 | S | - |
| 2 | IN0, Constant3 | - |
| 3 | IN0 | Constant3 |
| 4 | IN0 | Constant3, VarA |
| 5 | IN0 | Constant3, VarA, VarB |
| 6 | ADD2 | Constant3, VarA, VarB, IN0 |
| 7 | ADD2 | Constant3, VarA, VarB, IN0, VarE |
| 8 | ADD1 | Constant3, VarA, VarB, IN0, VarE, ADD2 |
| 9 | ADD1 | Constant3, VarA, VarB, IN0, VarE, ADD2, VarG |
| 10 | OUT0 | Constant3, VarA, VarB, IN0, VarE, ADD2, VarG, ADD1 |
| 11 | - | Constant3, VarA, VarB, IN0, VarE, ADD2, VarG, ADD1, OUT0 |

(b) Tabelle zur Sequenzialisierung

Abbildung A.1: Sequenzialisierung zu dem Beispiel aus Abbildung 4.19

# B Aufbau und Eigenschaften des Assemblercodes

Da im Rahmen dieser Arbeit die Syntax und Semantik des verwendeten Assemblercodes eine wichtige Rolle spielen, wird der Assemblercode mit seinen Eigenschaften in diesem Kapitel vorgestellt.
Sowohl der im ViSARD, als auch im LiSARD [DPZ+13], einem Softcore Prozessor auf LabVIEW-Basis, eingesetzte Assemblercode besitzt drei Operanden je Befehl:

- zwei Eingabeoperanden (auch Argumente genannt)
- ein Ergebnisoperand

Sollte ein Befehl weniger Argumente benötigen, oder kein Ergebnis erzeugen, so werden die entsprechenden Stellen durch Platzhalter aufgefüllt. Ein Platzhalter in diesem Assemblercode ist als „?“ definiert. Die Definitionen von Variablen und Konstanten sind nach demselben Muster aufgebaut. Im Folgenden werden in Abbildung B.1 Beispiele für mögliche Assemblercodebefehle dargestellt.

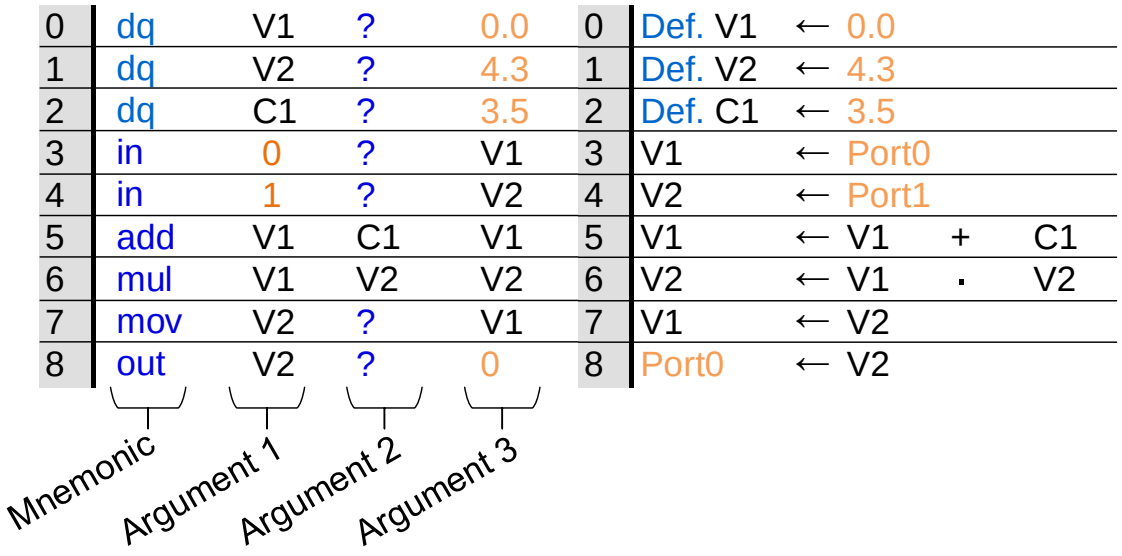



Abbildung B.1: Beispiele für ViSARD Assemblercode

Wie in Abbildung B.1 gesehen werden kann, beginnt jeder Befehl mit einer Operation (*Mnemonic*). In den ersten drei Zeilen des Beispiels werden mit dem Operator „dq“ drei Variablen definiert. Bei einer Definition von Variablen ist das erste Argument (*Argument 1*) immer der Name der Variable. Daraufhin wird diesem Namen mit dem letzten Argument (*Argument 3*) ein

Anfangswert zugeordnet. Da die Definition eines Namens inklusive Wertzuweisung ausreichen, wird keine weitere Information als Argument (*Argument 2*) benötigt und daher wird ein Platzhalter (*?*-Symbol) als Operand verwendet. Bei entsprechenden Definitionen ist es sowohl möglich, wie in Abbildung B.1 gezeigt, Gleitkommawerte anzugeben, als auch die Angabe von einem repräsentierenden Hexadezimalcode ist möglich. Dieser Hexadezimalcode muss dabei, je nach Konfiguration des ViSARD, entweder mit 32- oder 64-Bit nach dem Floating-Point Standard [IEE85] aufgebaut sein. Ein Beispiel wird in Zeile 0 von Abbildung B.2 dargestellt.

```
0 | dq      V1    ?       0x3FF0000000000000 ; 1
1 | ; Comments after „ ; “
```

Abbildung B.2: Beispiel für hexadezimale Variablen und Kommentare

Zusätzlich wird in Abbildung B.2 in Zeile 1 ein Beispiel für einen Kommentar gezeigt. Kommentare beginnen immer mit dem Schlüsselsymbol „ ; “. Das Symbol kann dabei sowohl zu Beginn einer Zeile, als auch nach einer Anweisung stehen. Sämtliche Zeichen ab dem Kommentarsymbol bis zum Ende der Zeile werden vom Assembler ignoriert.
Nach den Variablendefinitionen können beliebig viele Operationen ausgeführt werden. Exemplarisch ist in den Zeilen 3 bis 8 von Abbildung B.1 zu sehen, dass die Variablen *V1* und *V2* zunächst mit zwei von extern bestimmten Werten aus *Port0* bzw. *Port1* belegt werden. Die Anfangsbelegung dieser Variablen spielt dabei keine Rolle und wird überschrieben. In Zeile 5 und 6 sind exemplarisch eine Addition und eine Multiplikation als Vertreter der Operatoren mit zwei Eingabeargumenten und einem Ergebnis dargestellt. Zu beachten ist, wie bereits bei den externen Inputs, dass das Ergebnis immer in das dritte Argument (*Argument 3*) gespeichert wird. Die letzten beiden Befehle sind Vertreter der Operatoren mit je einem Eingabeargument und einem Ergebnis. Der Unterschied besteht darin, dass bei der letzten Operation (*out*) kein Ergebnis gespeichert, sondern an die dem Softcore umgebende Logik (*Port0*) weitergegeben wird.
Eine Liste der Obermenge aller Befehle, die in den Versuchsaufbauten der praktischen Versuche verwendet wurden, mit internen Aufbau ist in den Tabellen B.1 und B.2 zu finden.

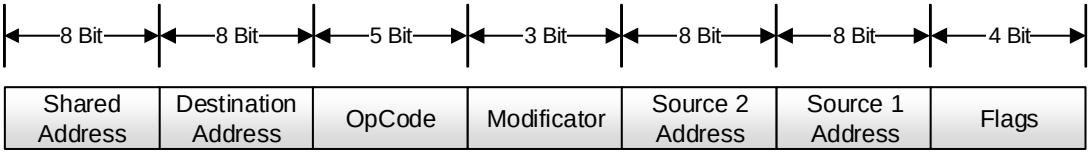



Abbildung B.3: Bitweiser Aufbau eines Maschinencodebefehls

In Abbildung B.3 wird der allgemeine Aufbau jedes Maschinenbefehls gezeigt, nachdem ein Assembler den Assemblercode übersetzt hat. Der Maschinencode jedes Befehls besteht unter anderem aus je einer 8-Bit breiten Speicheradresse, mit der bis zu 255 Variablen adressiert werden können: eine Adresse im geteilten Speicher (*Shared Address*), der Zieladresse des Ergebnisses (*Destination Address*) und zwei Quelladressen der jeweilig für eine Operation benötigten Argumente (*Source 1 Address* und *Source 2 Address*). Konflikte z. B. im geteilten Speicher (*Shared Address*) werden zur Designzeit im Assembler aufgelöst. Wird ein Ergebnis

geschrieben, so geschieht das parallel mit dem Kern-exklusiven Speicher, das Lesen wird über die Flags gesteuert. Weiterhin können die zur Verfügung stehenden Operatoren über einen 5-Bit breiten OpCode (*OpCode*) ausgewählt werden. Es werden drei Bits verwendet, um Einstellungen innerhalb der ALU vornehmen zu können (*Modificator*). Die Flags (*Flags*) werden zusätzlich zur Bypass-Nutzung verwendet.

Eine nicht unterstützte Art von Befehlen sind Sprungbefehle, die den Zeiger auf den nächsten Befehl bzw. Operator manipulieren, wie bereits ausführlich in Abschnitt 4.4.1 erläutert. Der Grund liegt in der Aufgabenklasse des Prozessors und der aus der Aufgabenklasse resultierenden Anforderung nach vollständiger Analysierbarkeit des Quellcodes zur Designzeit. Sprungbefehle, insbesondere bedingte Sprünge, würden eine Analyse verhindern, da diese erst zur Laufzeit aufgelöst werden können und damit eine Vorhersagbarkeit über die exakte Ausführungszeit des Assemblercodes bzw. des daraus resultierenden Maschinencodes verhindern würden. Der Grund im Verbot von Sprungbefehlen liegt, wie bereits erläutert, in der Notwendigkeit der vollständigen zeitlichen Analysierbarkeit des Assemblercodes. Diese Einschränkung ist allerdings ein unwesentlicher Nachteil, da im Prozessor eine Hardwareschleife umgesetzt wurde und etwaig weitere benötigte Schleifen innerhalb der jeweiligen Tools, wie dem in Abschnitt 5.2 vorgestellten Tool zur modellbasierten Assemblercodeerzeugung, Mechanismen umgesetzt wurden, um Schleifen zu definieren, die dann automatisiert in ausgerollten Assemblercode übersetzt werden.
In den folgenden Tabellen B.1 und B.2 steht die Spalte SGL/DBL für die Einsetzbarkeit des Mnemonic im Single Precision und/oder Double Precision-Modus des Softcores.

| Mnemonic | SGL/ DBL | Operand 1 | Operand 2 | Zieladresse | OpCode | Modifikator | Beschreibung |
|---|---|---|---|---|---|---|---|
| DQ | both | Variable Name | ? | Value | none | none | Variablendeklaration mit Initialwert |
| In | both | Input Address | ? | Variable Name | 00000 | 000 | Mux Eingabeadresse → Datenspeicher |
| Out | both | Variable Name | ? | Output Address | 00001 | 000 | Mux Datenspeicher → Ausgabeadresse |
| Mov | both | Variable Name | ? | Variable Name | 00010 | 000 | Op3 = Op1 |
| Abs | both | Variable Name | ? | Variable Name | 00010 | 100 | Op3 = \|Op1\| |
| Add | both | Variable Name | Variable Name | Variable Name | 00011 | 000 | Op3 = Op1 + Op2 |
| Sub | both | Variable Name | Variable Name | Variable Name | 00011 | 001 | Op3 = Op1 - Op2 |
| AbsSub | both | Variable Name | Variable Name | Variable Name | 00011 | 111 | Op3 = \|Op1\| - \|Op2\| |
| Mul | both | Variable Name | Variable Name | Variable Name | 00101 | 000 | Op3 = Op1 · Op2 |
| Div | both | Variable Name | Variable Name | Variable Name | 00110 | 000 | Op3 = Op1 ÷ Op2 |
| WbNone | both | ? | ? | ? | 00111 | 000 | NOP-Befehl |
| Sqrt | both | Variable Name | ? | Variable Name | 01000 | 000 | Op3 = $\sqrt{Op1}$ |
| NatExp | both | Variable Name | ? | Variable Name | 11010 | 000 | Op3 = $e^{Op1}$ |
| Sin | SGL | Variable Name | ? | Variable Name | 01001 | 000 | Op3 = sin(Op1) |
| Cos | SGL | Variable Name | ? | Variable Name | 01010 | 000 | Op3 = cos(Op1) |
| Sin | DBL | Variable Name | Select | Variable Name | 01001 | 000 | Op3 = sin(Op1) |
| Cos | DBL | Variable Name | Select | Variable Name | 01010 | 000 | Op3 = cos(Op1) |
| SGLtoDBL | DBL | Variable Name | ? | Variable Name | 01011 | 000 | Konvertierung Single → Double |
| DBLtoSGL | DBL | Variable Name | ? | Variable Name | 01111 | 000 | Konvertierung Double → Single |
| SGLtoI16 | SGL | Variable Name | ? | Variable Name | 01100 | 000 | Konvertierung Single → Int16 |
| I16toSGL | SGL | Variable Name | ? | Variable Name | 01101 | 000 | Konvertierung Int16 → Single |
| DBLtoI32 | DBL | Variable Name | ? | Variable Name | 01100 | 000 | Konvertierung Double → Int32 |
| I32toDBL | DBL | Variable Name | ? | Variable Name | 01101 | 000 | Konvertierung Int32 → Double |
| DBLtoI16 | DBL | Variable Name | ? | Variable Name | 11001 | 000 | Konvertierung Double → Int16 |
| I16toDBL | DBL | Variable Name | ? | Variable Name | 11000 | 000 | Konvertierung Int16 → Double |
| UI8toDBL | DBL | Variable Name | ? | Variable Name | 11011 | 000 | Konvertierung UInt8 → Double |
| DBLtoUI8 | DBL | Variable Name | ? | Variable Name | 11100 | 000 | Konvertierung Double → UInt8 |

Tabelle B.1: Mnemonic und OpCode Tabelle Teil 1

| Mnemonic | SGL/ DBL | Operand 1 | Operand 2 | Zieladresse | OpCode | Modifikator | Beschreibung |
|---|---|---|---|---|---|---|---|
| MovIs Positive | both | Variable Name | Variable Name | Variable Name | 10000 | 000 | Op3 = Op1 if sign(Op2) = 0 |
| MovIs Negative | both | Variable Name | Variable Name | Variable Name | 10001 | 000 | Op3 = Op1 if sign(Op2) = 1 |
| MovIs Null | both | Variable Name | Variable Name | Variable Name | 10010 | 000 | Op3 = Op1 if significand(Op2) = 0 und exponent(Op2) = 0 |
| MovIsn Null | both | Variable Name | Variable Name | Variable Name | 10011 | 000 | Op3 = Op1 if significand(Op2) $\neq$ 0 und exponent(Op2) = 0 |
| MovIs NaN | both | Variable Name | Variable Name | Variable Name | 10100 | 000 | Op3 = Op1 if significand(Op2) $\neq$ 0 und exponent(Op2) is all 0 |
| MovIsn NaN | both | Variable Name | Variable Name | Variable Name | 10101 | 000 | Op3 = Op1 if significand(Op2) = 0 oder exponent(Op2) is not all 1 |
| MovIs INF | both | Variable Name | Variable Name | Variable Name | 10110 | 000 | Op3 = Op1 if significand(Op2) = 0 und exponent(Op2) is all 1 |
| MovIsn INF | both | Variable Name | Variable Name | Variable Name | 10111 | 000 | Op3 = Op1 if significand(Op2) $\neq$ 0 oder exponent(Op2) is not all 1 |

Tabelle B.2: Mnemonic und OpCode Tabelle Teil 2

# C Details zur Implementierung des ViSARD Softcore Prozessor

## C.1 Der Sinus/Cosinus-Operator

Im Rahmen der Auswahl geeigneter EUs für die Experimente der partiellen Rekonfiguration innerhalb der ALU wurde der ViSARD um verschiedene Optionen zur Berechnung des Sinus und Cosinus in der doppelten Genauigkeit erweitert. Zuvor waren diese Operatoren nur in der Konfiguration der einfachen Genauigkeit (32-Bit) verfügbar. Dabei wurde der von Xilinx zur Verfügung gestellte „Coordinate Rotational Digital Computer IP Code“ (Cordic) [Xil17a] verwendet, da der für die anderen EUs verwendete Floating Point IP Core diese Funktionalität nicht bietet. Der Cordic erlaubt dabei Berechnungen von Sinus und Cosinus mit einer Genauigkeit von bis zu 48 Bit im Fixed-Point Zahlenformat.
Da der Softcore allerdings nur mit Floating-Point Formaten arbeiten kann, wurde vor bzw. nach dem Cordic IP Core jeweils ein Konvertierungs-IP eingesetzt, um die Zahlenformate ineinander umzurechnen.
Bei dem Ganzzahlformat (Fixed-Point) wird die Variable dabei in zwei Integer Zahlen dargestellt: einem ganzzahligen Teil und einer Zahl zur Darstellung des Nachkommaanteils. Im Fall des Cordic bedeutet das immer ein ganzzahliger Anteil von 3 Bit als Eingabe und 2 Bit als Ausgabe, mit einem variablen Nachkommaanteil. Entsprechend dem Maximum von insgesamt 48 Bit, wurde also ein Anteil von 45 Bit Nachkommagenauigkeit als Eingabe und 46 Bit Nachkommagenauigkeit als Ausgabe gewählt.
Der Operator kann aktuell in zwei Konfigurationen verwendet werden:

- Paralleler Modus
  In diesem Modus ist der Operator voll gepipelined, wie jeder andere Operator im ViSARD ebenfalls. Der Nachteil hierbei ist, dass die maximale Genauigkeit 40 Bit beträgt, und das der Operator extrem viele Ressourcen (6183 LUTs, 5822 Flip-Flops, siehe Tabelle C.1) benötigt.

- Serieller Modus
  Dieser Modus verwendet keine interne Operator-Pipeline. Das bedeutet, dass nur eine Berechnung durchgeführt werden kann und alle weiteren Berechnungen von Sinus/Cosinus-Werten auf das Ergebnis dieser Berechnung warten müssen. Der Vorteil liegt in der größeren Genauigkeit von 48 Bit und in einem vergleichsweise geringen Ressourcenverbrauch (1352 LUTs, 1181 Flip-Flops, siehe Tabelle C.1)

Für die Experimente mit der partiellen Rekonfiguration wurde der Sinus/Cosinus-Operator im seriellen Modus verwendet, wie in Abbildung C.1 dargestellt.

Wie in Abbildung C.1 gesehen werden kann, werden sowohl der Sinus, als auch der Cosinus parallel ausgegeben. Da der ViSARD allerdings in einem Takt lediglich ein Ergebnis speichern

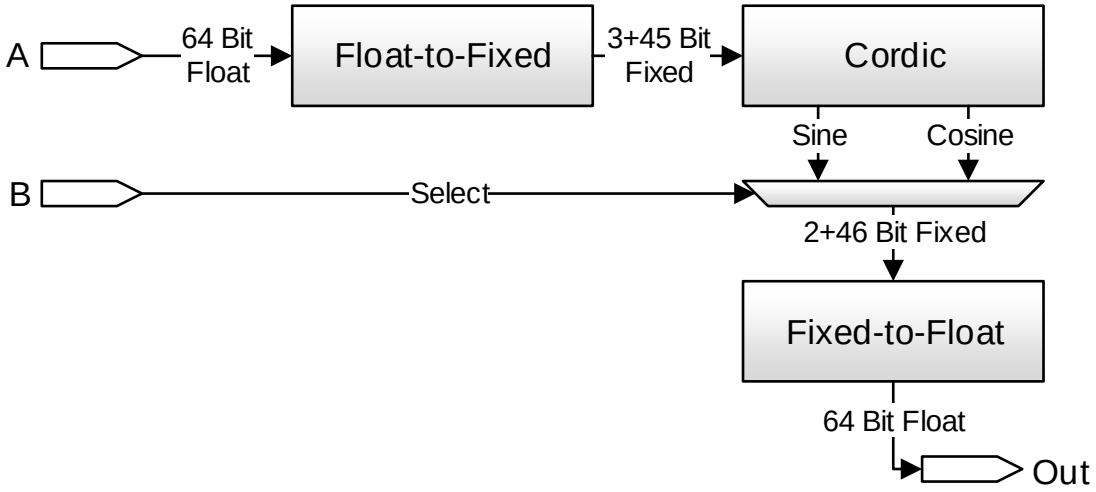



Abbildung C.1: Aufbau des Sinus/Cosinus-Operators im ViSARD

kann, wird über das zur Verfügung stehende zweite Argument ($B$) bei Eingabe des ersten Arguments ($A$) bereits festgelegt, welches der Ergebnisse an die ALU weitergegeben wird.

## C.2 Der partielle Rekonfigurationscontroller

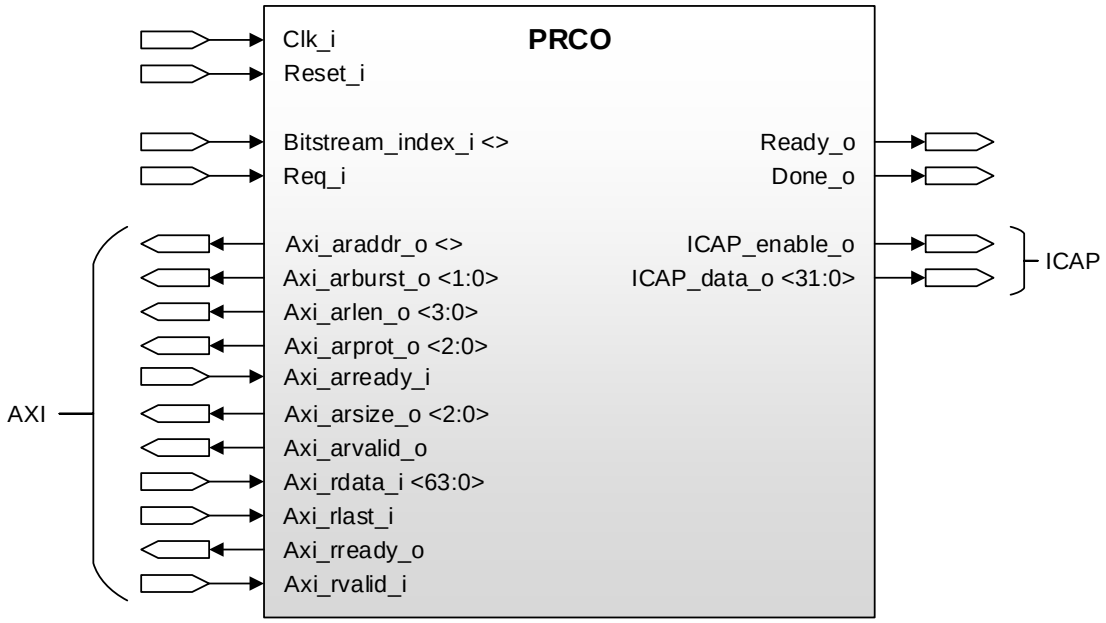



Abbildung C.2: Interface des PRCO IP-Blocks

## C.3 Ressourcenbedarf der rekonfigurierbaren Operatoren

In Tabelle C.1 werden alle zur partiellen Rekonfiguration des ViSARD in Betracht gezogenen EUs, mit ihren jeweiligen Konfigurationen und Ressourcenbedarf aufgelistet.
Die Ressourcenkonfiguration des BRAMs können dabei die Werte: Ohne und Voll, und die DSP Ressourcenkonfiguration die Werte: Ohne, Mittel, Voll und Max annehmen. Diese Werte stehen für die Menge an verwendeten Ressourcen des jeweiligen Typs innerhalb dieses IP-Cores.

| EU | Konfiguration | Latenz | LUTs | FFs | BRAMs | DSPs |
|---|---|---|---|---|---|---|
| Add & Sub | DSP: Ohne | 4 | 638 | 350 | 0 | 0 |
| Add & Sub | DSP: Ohne | 12 | 650 | 1063 | 0 | 0 |
| Add & Sub | DSP: Voll | 4 | 614 | 948 | 0 | 3 |
| Multiplikation | DSP: Ohne | 4 | 2657 | 892 | 0 | 0 |
| Multiplikation | DSP: Ohne | 9 | 2241 | 2419 | 0 | 0 |
| Multiplikation | DSP: Mittel | 15 | 273 | 559 | 0 | 9 |
| Multiplikation | DSP: Voll | 15 | 220 | 498 | 0 | 10 |
| Multiplikation | DSP: Max | 16 | 202 | 490 | 0 | 11 |
| Division | - | 30 | 3196 | 3018 | 0 | 0 |
| Division | - | 40 | 3251 | 3026 | 0 | 0 |
| Wurzelberechnung | - | 28 | 1713 | 1654 | 0 | 0 |
| Wurzelberechnung | - | 57 | 1730 | 3230 | 0 | 0 |
| Sin & Cosinus | Seriell, 32 Bit Genauigkeit | 41 | 928 | 548 | 0 | 0 |
| Sin & Cosinus | Seriell, 40 Bit Genauigkeit | 50 | 1159 | 682 | 0 | 0 |
| Sin & Cosinus | Seriell, 48 Bit Genauigkeit | 57 | 1352 | 1181 | 0 | 0 |
| Sin & Cosinus | Parallel, 40 Bit Genauigkeit | 47 | 6183 | 5822 | 0 | 0 |
| nat. Exponential-funktion | BRAM: Ohne, DSP: Ohne | 20 | 6061 | 3979 | 0 | 0 |
| nat. Exponential-funktion | BRAM: Ohne, DSP: Mittel | 20 | 2096 | 1112 | 0 | 15 |
| nat. Exponential-funktion | BRAM: Ohne, DSP: Voll | 20 | 1853 | 1022 | 0 | 26 |
| nat. Exponential-funktion | BRAM: Voll, DSP: Ohne | 20 | 5209 | 3702 | 2,5 | 0 |
| nat. Exponential-funktion | BRAM: Voll, DSP: Mittel | 20 | 1122 | 1029 | 2,5 | 15 |
| nat. Exponential-funktion | BRAM: Voll, DSP: Voll | 20 | 828 | 919 | 2,5 | 26 |
| nat. Exponential-funktion | BRAM: Voll, DSP: Voll | 57 | 831 | 1953 | 2,5 | 26 |

Tabelle C.1: Ressourcenverbrauch der zur Rekonfiguration relevanten EUs

Bei allen Sinus und Cosinus Operatoren wird dabei das sogenannte „Radian Phase Format“ und keine Coarse Rotation verwendet. Das Ergebnis wird dann auf die am nächsten liegende natürliche Zahl gerundet.

## C.4 Grafische Ergebnisse der Rekonfigurationszeiten

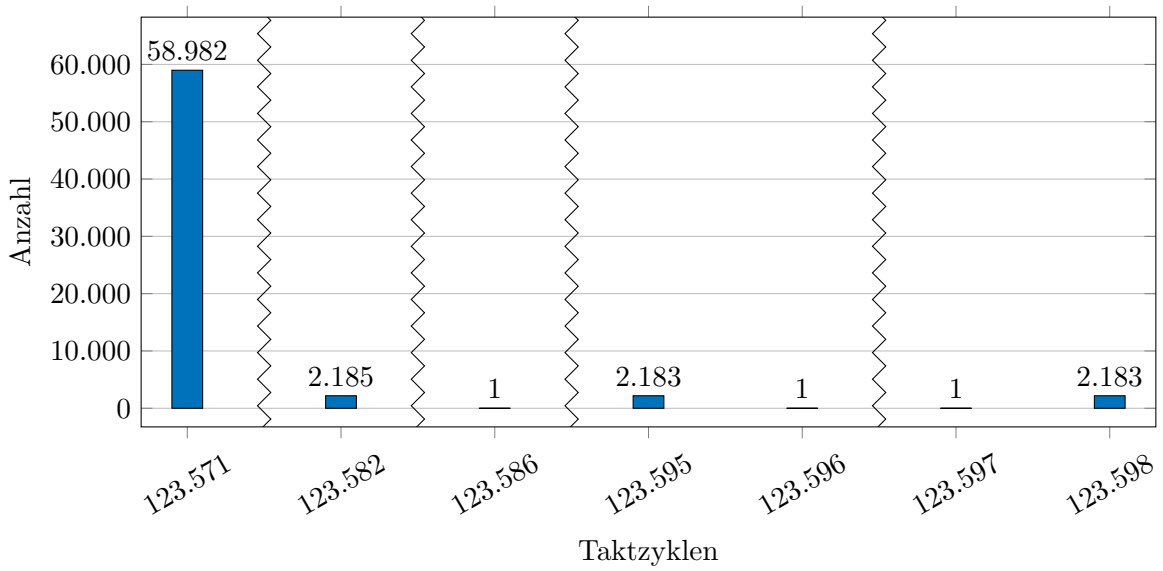



Abbildung C.3: Rekonfigurationszeiten Multi-EU Rekonfiguration (Xilinx Standard Bitstream)

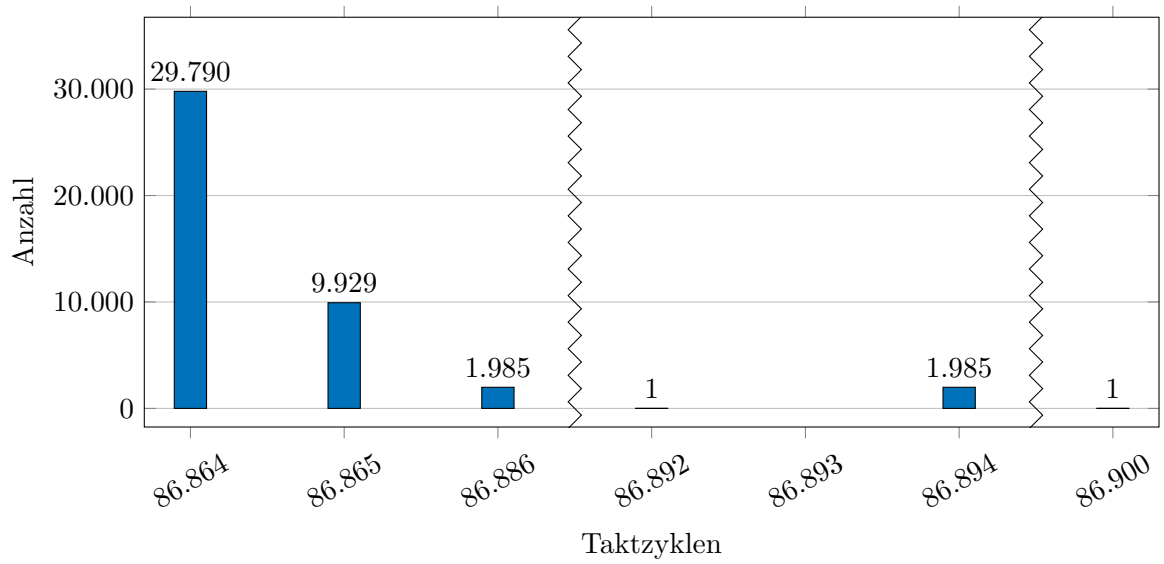



Abbildung C.4: Rekonfigurationszeiten Multi-EU Rekonfiguration (Xilinx komprimierter Bitstream)

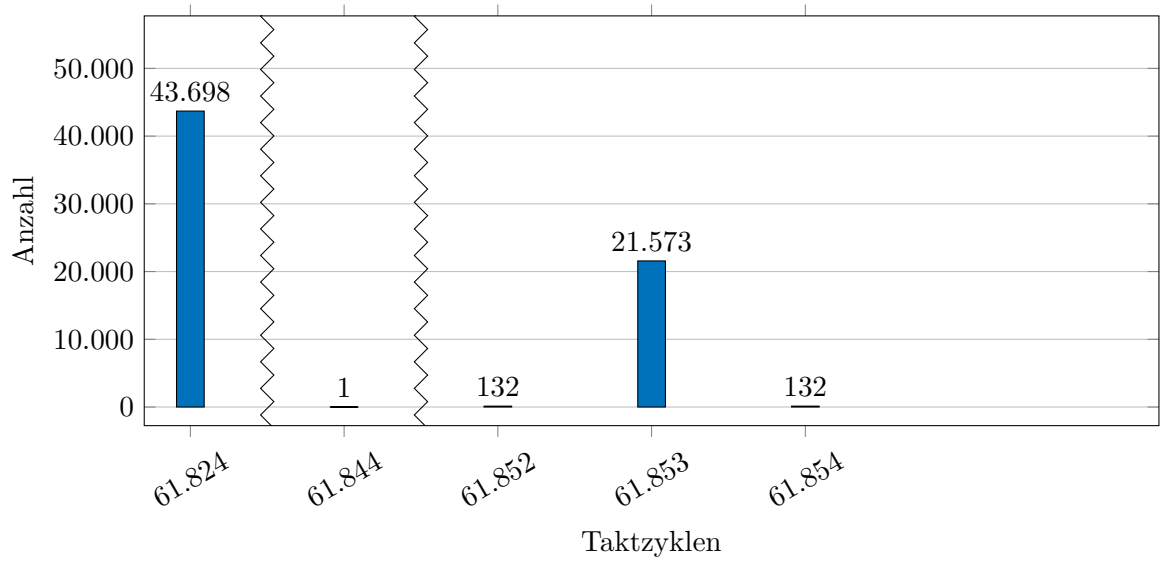



Abbildung C.5: Rekonfigurationszeiten Multi-EU Rekonfiguration (torCombitgen Bitstream)

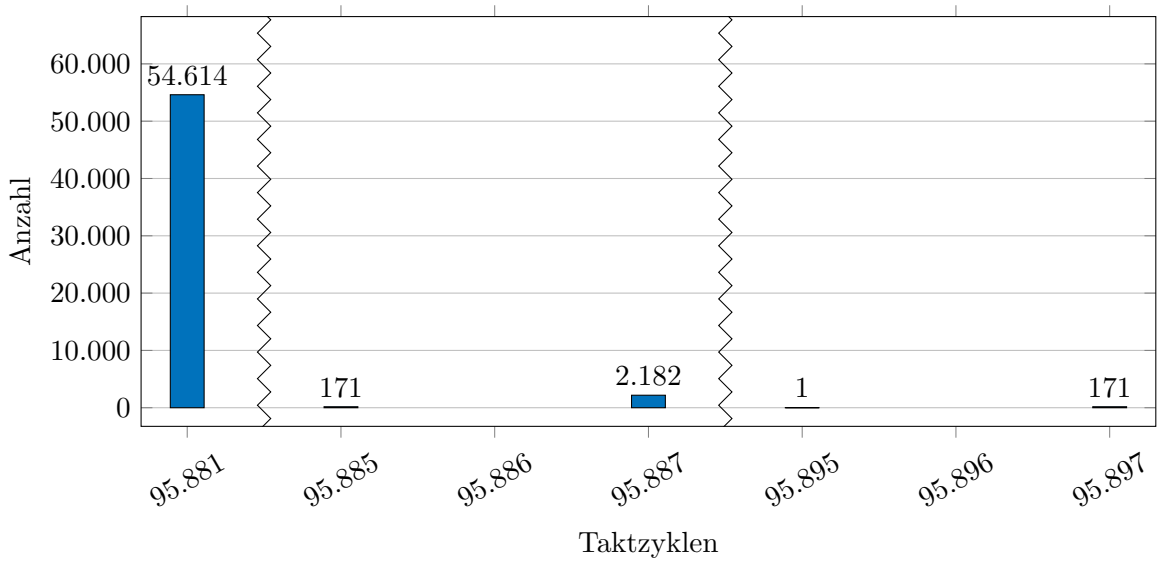



Abbildung C.6: Rekonfigurationszeiten kompletter Softcore (Xilinx Standard Bitstream)

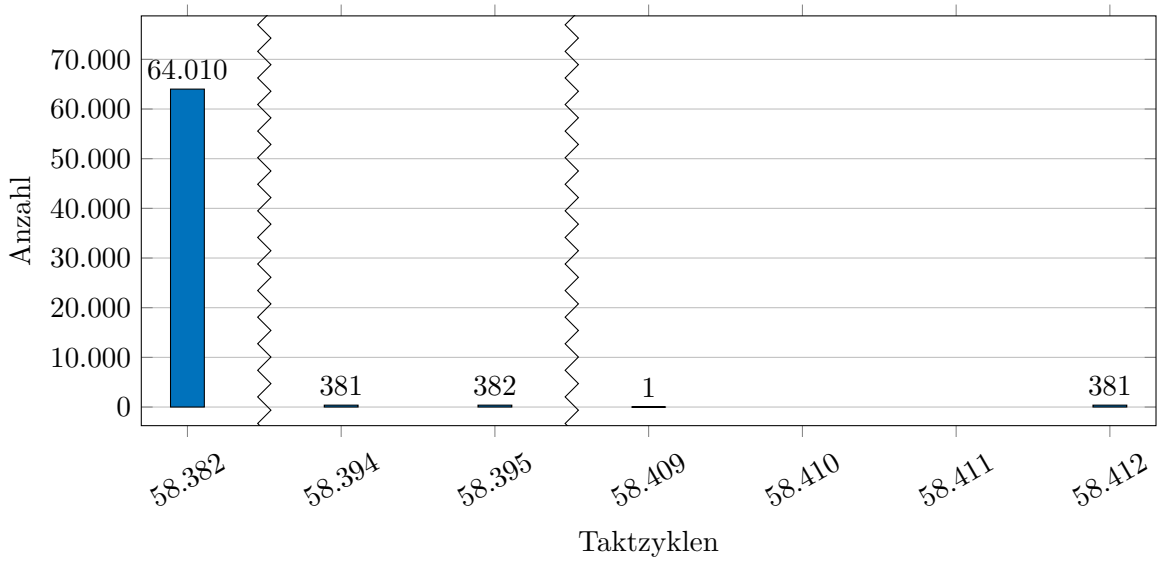



Abbildung C.7: Rekonfigurationszeiten kompletter Softcore (Xilinx komprimierter Bitstream)

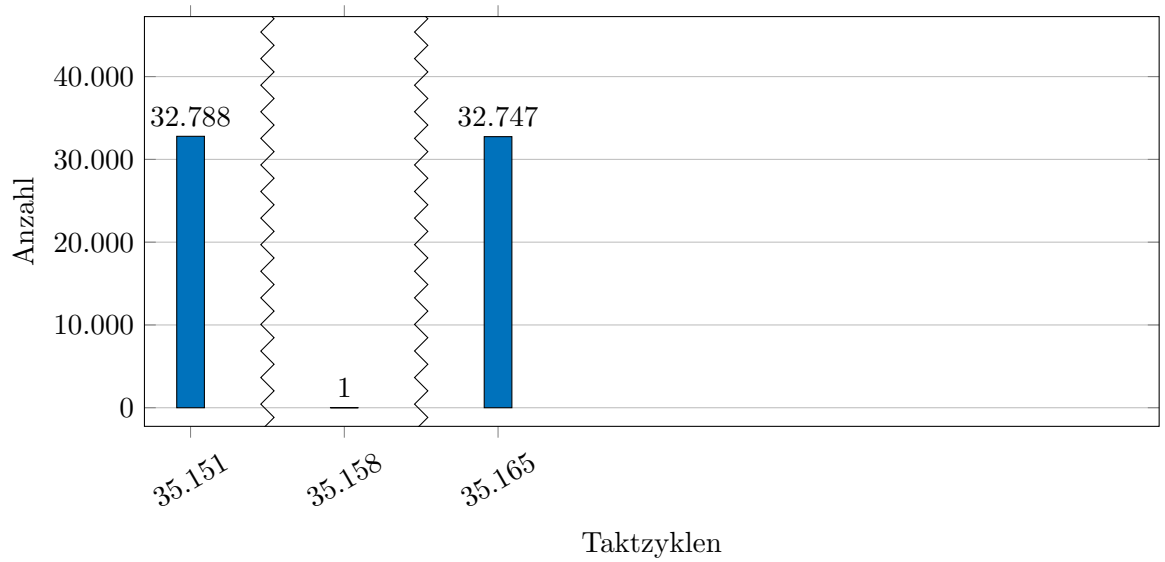



Abbildung C.8: Rekonfigurationszeiten kompletter Softcore (torCombitgen Bitstream)

# D Das MACG-Tool

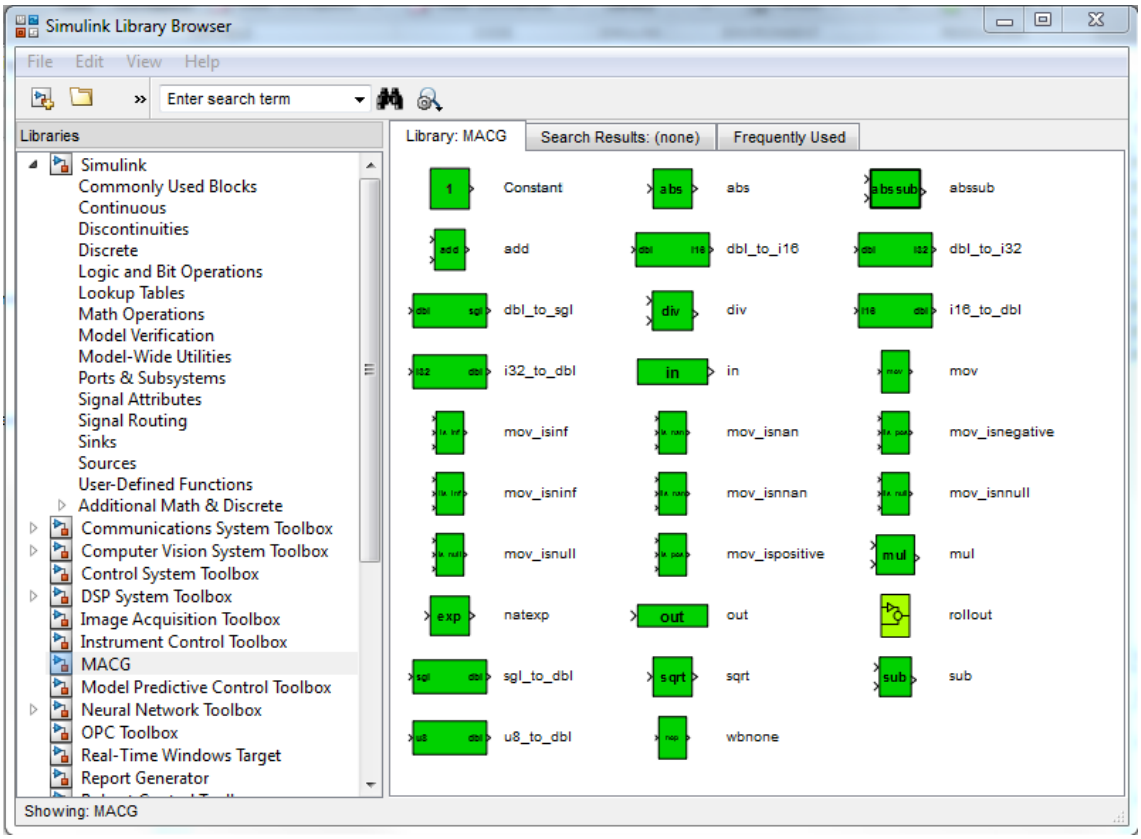



Abbildung D.1: MACG Bibliothek

Mit Hilfe dieser Bibliothek lassen sich beliebig komplexe Datenflussmodelle erstellen, zu Subsystemen zusammenfassen und in weiteren Datenflussmodellen wiederverwenden.

## D.1 Beispiel Assemblercode

Zu sehen sind: Kopfzeilen mit Modell- und Optimierungsinformationen (*Header*), der Variablen- und Konstantendeklarationsabschnitt (*Declaration*) sowie der Algorithmusabschnitt (*Algorithm*). Im ersten Abschnitt werden Name des Datenflussmodells, Erstellungsdatum, Name des Autors, verwendete Schedulingstrategie und die verwendeten Optimierungen abgespeichert.
Daraufhin werden alle notwendigen Konstanten sowie verwendete Variablen deklariert. Begonnen wird dabei immer mit einer „Dummy“-Variable, auf die nicht oder falsch initialisierter Programmcode des ViSARD Softcore Prozessors während der Ausführung schreiben kann. Im Anschluss wird eine Konstante mit dem Wert eins (*Constant1*), gefolgt von vier Variablen,

```
1  ; System: Dissertation_Example from 22-Aug-2018 13:58:23
2  ; Creator: M.Sc. Michael Kirchhoff
3  ; Streategy: Rollout_Priority; Optimizing: 1; Optimizations: 0000000
4  ;
5  ;Declare all Variables
6  dq avoid ? 0x0000000000000000 ; uninitialised program memory here
7  dq Constant1 ? 0x3FF0000000000000 ; 1
8  dq t_0 ? 0xffffffffffffffff ;
9  dq t_1 ? 0xffffffffffffffff ;
10 dq t_2 ? 0xffffffffffffffff ;
11 dq t_3 ? 0xffffffffffffffff ;
12 in 0 ? t_0 ;
13 in 1 ? t_1 ;
14 add Constant1 t_0 t_2 ;
15 mul t_1 t_2 t_3 ;
16 out t_3 ? 0 ;
```

Header

Declaration

Algorithm

Abbildung D.2: Beispielhafter Assemblercode

deklariert.
Im dritten Abschnitt sind die eigentlichen Assemblerbefehle aufgelistet. In diesem Beispiel werden zwei externe Werte aus Port 0 und Port 1 abgefragt und jeweils in einer Variable abgespeichert (*t_0* und *t_1*). Im Folgenden wird der erste Eingabewert inkrementiert und mit diesem Ergebnis eine Multiplikation mit dem zweiten Eingabewert ausgeführt. Das Resultat der Multiplikation wird mittels des letzten Befehls an die umgebende Logik ausgegeben.

## D.2 Python Assemblercode Simulator

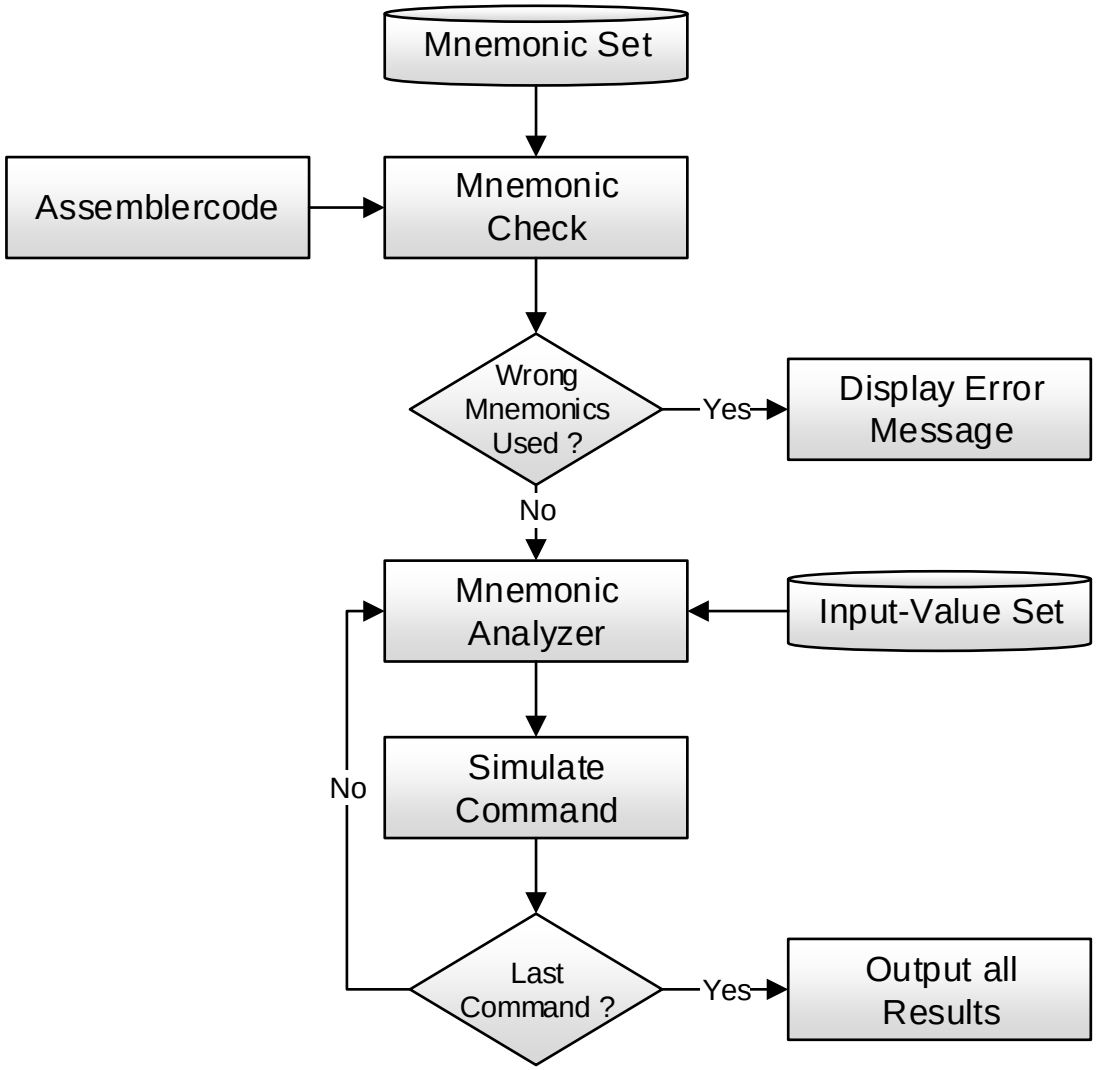



Abbildung D.3: Python Assemblercode Simulator

Zu sehen ist der in Python realisierte Assemblercodesimulator. Das Kommandofenster-Tool benötigt im einfachsten Fall lediglich den Pfad der zu testenden Assemblerdatei, rechnet alle Operationen sequentiell in der jeweilig definierten Reihenfolge und erzeugt für jeden im Code definierten „Out“-Befehl eine entsprechende Ausgabe. Zusätzlich werden Statistiken ausgegeben, wie die Anzahl an Codezeilen. Sollten externe Eingaben notwendig sein, oder andere Operationen verwendet werden, so können diese in einer dafür vorgesehenen Datei im Vorfeld definiert und anschließend verwendet werden.

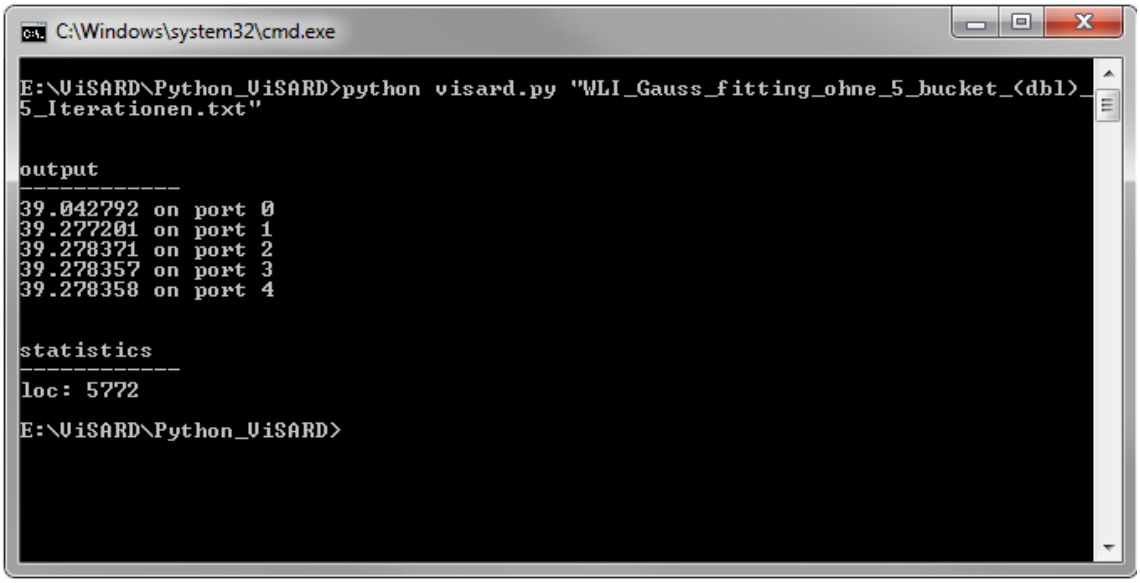



Abbildung D.4: Kommandozeilenausgabe des Python Assemblercode Simulator

## D.3 Testkonfigurationen

Innerhalb des MACG Tests wurden für den Vergleichswert in einem ersten Durchlauf sowohl graph- als auch assemblercodebasierte Optimierungen ausgeschaltet. In einem weiteren Durchlauf wurden die angegebenen Schedulingstrategien (meist Rollout Priority) in Kombination mit allen zur Verfügung stehenden graph- und assemblercodebasierten Optimierungen aktiviert.
Der verwendete Assembler hat dabei folgende Einstellungen verwendet:

- 10 Bit Adressbreite
- Schedulingverfahren FCFS (First Come First Served)
- Angenommene Softcore Rechengeschwindigkeit von 100 MHz

Dabei hatten die Operatoren die in Tabelle D.1 dargestellten Berechnungszeiten.

| Operation | Rechenzeit (Takte) |
|---|---|
| In | 0 |
| OUT | 0 |
| Abs | 0 |
| Mov | 0 |
| MovIsPositive | 0 |
| MovIsNegative | 0 |
| MovIsNull | 0 |
| MovIsnNull | 0 |
| MovIsNan | 0 |
| MovIsnNan | 0 |
| MovIsInf | 0 |
| MovIsnInf | 0 |
| ADD | 4 |
| SUB | 4 |
| AbsSub | 4 |
| MUL | 15 |
| DIV | 40 |
| SQRT | 29 |
| NatExp | 20 |
| DBLtoI32 | 3 |
| I32toDBL | 3 |
| SGLtoDBL | 2 |
| DBLtoSGL | 2 |
| I16toDBL | 3 |
| DBLtoI16 | 3 |
| UI8toDBL | 2 |
| DBLtoUI8 | 2 |

Tabelle D.1: Verwendete Operatortakte im Assembler

# E Das ViSARD-Assembler Tool

## E.1 Schedulingausdrücke

```
<Penalty>
  <Name> FCFS </Name>
  <Description> First Come First Served </Description>
  <Expression> 1 </Expression>
</Penalty>
<Penalty>
  <Name> SJF </Name>
  <Description> Shortest Job First </Description>
  <Expression> Instruction.Delay </Expression>
</Penalty>
<Penalty>
  <Name> LJF </Name>
  <Description> Longest Job First </Description>
  <Expression> 0-Instruction.Delay </Expression>
</Penalty>
<Penalty>
  <Name> EmptyMemory </Name>
  <Description> Prioritizes instructions that will
                free up Memory space </Description>
  <Expression> (Operand1.ReadOperationsLeft
                +Operand2.ReadOperationsLeft
                +5*Operand1.IsNoVariable+
                +5*Operand2.IsNoVariable)
                *20-Instruction.Delay
                -(Operand1.Availability.Bypass
                +Operand2.Availability.Bypass)
                *50 </Expression>
</Penalty>
<Penalty>
  <Name> RROmax </Name>
  <Description> Prioritize instructions with results that are
                required often </Description>
  <Expression> Instruction.Delay-
               100*Operand3.ReadOperationsTotal  </Expression>
</Penalty>
```

Zu sehen sind die vordefinierten Schedulingausdrücke:

- First Come First Served
  Das Scheduling ergibt sich einfach aus den startbaren Befehlen in der Reihenfolge, wie diese im zugrunde liegenden Assemblercode definiert sind.
- Shortest Job First
  Aus der Liste an startbaren Befehlen wird immer der Befehl ausgewählt, der die kürzeste Rechenzeit benötigt.
- Longest Job First
  Das Gegenstück zum „Shortest Job First"-Scheduling: Aus der Liste an startbaren Befehlen wird immer der Befehl ausgewählt, der die längste Rechenzeit benötigt.
- EmptyMemory
  Hier werden Operationen bevorzugt, nach deren Abarbeitung möglichst Speicherzellen zur Überschreibung frei werden.
- RROmax
  Hier werden Operationen bevorzugt, deren Ergebnisse von vielen folgenden Operationen benötigt werden. Also Operationen, von denen sehr viele weitere Operationen abhängig sind.

## E.2 Testkonfigurationen

Die Assemblereinstellungen, die für das Benchmarking verwendet wurden, entsprechen den aktuellen Einstellungen die für den ViSARD Softcore Prozessor verwendet werden und werden im folgenden im Detail aufgeführt:

- 8 Bit Adressbreite
- Schedulingverfahren RROmax (Es werden Operationen bevorzugt, deren Ergebnisse frequentiert benötigt werden)
- Assemblercode wird übersetzt für ein-Prozessorkern Softcore Konfiguration
- Softcore Rechengeschwindigkeit von 100 MHz

Es gelten dieselben Rechenzeiten wie schon in Tabelle D.1 vorgestellt. Das Benchmarking-Tool hat die folgenden Einstellungen verwendet:

- Assemblercodelänge: 100-1000 LoC (Lines of Code)
- Erhöhung der Assemblercodelänge je neu erzeugter Datei: 100 LoC
- Testdurchläufe mit Abhängigkeitsgraden von 1 bis 350 in 10er Schritten
- 100 Durchläufe je Abhängigkeitsgrad und Codelänge
- Maximale Anzahl an Variablen und Konstanten je Assemblercode: 100
- Jede Variable wird mindestens einmal gelesen, bevor diese erneut geschrieben wird
- Wahrscheinlichkeitsverteilung der Operationen:

| Operation | Wahrscheinlichkeit (%) |
|---|---|
| ADD | 30% |
| SUB | 20% |
| MUL | 20% |
| DIV | 10% |
| „Rest[1]“ | 20% |

Tabelle E.1: Wahrscheinlichkeitsverteilung der Operatoren

[1]Unter „Rest“ sind alle zuvor nicht aufgezählten Operationen mit zu gleichen Teilen aufgeteilter Wahrscheinlichkeit zu verstehen.

# Abkürzungsverzeichnis

| | |
|---|---|
| **ALU** | Arithmetic Logic Unit |
| **ASIC** | Application-Specific Integrated Circuit |
| **BRAM** | Block Random-Access Memory |
| **CAP** | Configuration Access Port |
| **CLB** | Configurable Logic Block |
| **CPI** | Average Clock Cycles per Instruction |
| **CPU** | Central Processing Unit |
| **DMA** | Direct Memory Access |
| **DPM** | Dynamic Power Management |
| **DSP** | Digital Signal Processor |
| **DVS** | Dynamic Voltage Scaling |
| **EDMA** | Enhanced DMA |
| **eMMC** | Embedded Multi Media Card |
| **EU** | Execution Unit |
| **FF** | Flip-Flop |
| **FiFo** | First-in First-Out |
| **FPGA** | Field Programmable Gate Array |
| **FPU** | Floting Point Unit |
| **FSM** | Finite State Machine |
| **GPIO** | General Purpose Input/Output |
| **GPU** | Graphic Processing Unit |
| **HDL** | Hardware Description Language |
| **HLL** | High Level Language |
| **ICAP** | Internal Configuration Access Port |
| **IP** | Intellectual Property |
| **IOB** | Input/Output Block |
| **PRC** | Partial Reconfiguration Componentent |

| | |
|---|---|
| **LoC** | Lines of Code |
| **LUT** | Look-Up Table |
| **MACG** | Modelbased Assembly Code Generator |
| **MDSD** | Model-Driven Software Development |
| **MIMD** | Multiple Instruction Multiple Data |
| **MPMC** | Multi-Port Memory Controller |
| **NCD** | Native Circuit Description |
| **NGC** | Native Generic Constraint |
| **NGD** | Native Generic Database |
| **PCAP** | Processor Configuration Access Port |
| **PLA** | Programmable Logic Array |
| **PLB** | Processor Local Bus |
| **PR** | Partial Reconfiguration |
| **PRC** | Partial Reconfiguration Component |
| **PRCO** | Partial Reconfiguration Controller |
| **PROM** | Programmable Read-Only Memory |
| **QSPI** | Quad Serial Peripheral Interface |
| **ROM** | Read-Only Memory |
| **RT** | Register Transfer |
| **RTL** | Register Transfer Level |
| **SIMD** | Single Instruction Multiple Data |
| **SiP** | System-in-Package |
| **SoB** | System-on-Board |
| **SoC** | System-on-Chip |
| **SoM** | System-on-Module |
| **SoRC** | System-on-Reprogrammable-Chip |
| **TDP** | Thermal Design Power |
| **TDR** | Time-Distance Ratio |
| **TTM** | Time to Market |
| **ViSARD** | VHDL Integrated Softcore Architecture for Reconfigurable Devices |
| **VLIW** | Very Long Instruction Word |
| **VLSI** | Very Large Scale Integration |

**WLI** White Light Interferometry

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Literaturverzeichnis


[AE06] Arbinger, Don; Erdmann, Jeremy: Designing with an embedded soft-core processor. In: *Embedded cracking the code to systems development* (2006)

[Aho08] Aho, Alfred V.: *Compiler: Prinzipien, Techniken und Werkzeuge.* Pearson Deutschland GmbH, 2008 (Pearson Studium Informatik). – ISBN 978–3–8273–7097–6

[Amt10] Amthor, Arvid: *Modellbasierte Regelung von Nanopositionier-und Nanomessmaschinen.* VDI-Verlag, 2010 (Fortschritt-Berichte VDI: Reihe 8, Mess-, Steuerungs- und Regelungstechnik). – ISBN 978–3–18–517908–2

[Ash07] Ashenden, Peter J.: *Digital Design (VHDL): An Embedded Systems Approach Using VHDL.* Elsevier, 2007. – ISBN 978–0–12–369528–4

[Bae10] Baer, Jean-Loup: *Microprocessor architecture: from simple pipelines to chip multiprocessors.* Cambridge University Press, 2010. – ISBN 978–0–521–76992–1

[Bap14] Bapat, Ravindra B.: *Graphs and matrices.* Springer, 2014. – ISBN 978–1–4471–6569–9

[BC12] Barry, Peter; Crowley, Patrick: *Modern Embedded Computing: Designing Connected, Pervasive, Media-Rich Systems.* Elsevier, 2012. – ISBN 978–0–12–391490–3

[BDM09] Blake, Geoffrey; Dreslinski, Ronald G. ; Mudge, Trevor: A survey of multicore processors. In: *IEEE Signal Processing Magazine* 26 (2009), Nr. 6. `http://dx.doi.org/10.1109/MSP.2009.934110`. – DOI 10.1109/MSP.2009.934110. – ISSN 1053–5888

[Ben83] Benington, H. D.: Production of Large Computer Programs. In: *Annals of the History of Computing* 5 (1983), Oct, Nr. 4, S. 350–361. `http://dx.doi.org/10.1109/MAHC.1983.10102`. – DOI 10.1109/MAHC.1983.10102. – ISSN 0164–1239

[BH03] Becker, Jürgen; Hartenstein, Reiner: Configware and Morphware going Mainstream. In: *Journal of Systems Architecture* 49 (2003), S. 127–142. `http://dx.doi.org/10.1016/S1383-7621(03)00073-0`. – DOI 10.1016/S1383–7621(03)00073–0. – ISSN 1383–7621

[BKT14] Beckhoff, Christian; Koch, Dirk ; Torresen, Jim: Portable module relocation and bitstream compression for Xilinx FPGAs. In: *24th International Conference on Field Programmable Logic and Applications (FPL)* IEEE, 2014. – ISSN 1946–147X, S. 1–8

[Boe88] Boehm, B. W.: A spiral model of software development and enhancement. In: *Computer* 21 (1988), May, Nr. 5, S. 61–72. http://dx.doi.org/10.1109/2.59. – DOI 10.1109/2.59. – ISSN 0018–9162

[BT75] Basil, Victor R.; Turner, Albert J.: Iterative enhancement: A practical technique for software development. In: *IEEE Transactions on Software Engineering* (1975), Nr. 4, S. 390–396. http://dx.doi.org/10.1109/TSE.1975.6312870. – DOI 10.1109/TSE.1975.6312870. – ISSN 0098–5589

[BVK08] Bunse, Christian; Von Knethen, Antje: *Vorgehensmodelle kompakt*. Spektrum Akademischer Verlag, 2008 (IT kompakt). – ISBN 978–3–8274–1950–7

[CH06] Czarnecki, Krzysztof; Helsen, Simon: Feature-based survey of model transformation approaches. In: *IBM Systems Journal* 45 (2006), Nr. 3, S. 621–645. http://dx.doi.org/10.1147/sj.453.0621. – DOI 10.1147/sj.453.0621. – ISSN 0018–8670

[CH11] Cofer, R.C.; Harding, Benjamin F.: *Rapid System Prototyping with FPGAs: Accelerating the Design Process*. Elsevier, 2011 (Embedded Technology). – ISBN 978–0–08–045737–6

[Cla11] Claus, Christopher: *Zum Einsatz dynamisch rekonfigurierbarer eingebetteter Systeme in der Bildverarbeitung*, Technische Universität München, Diss., 2011

[CLHH10] Chen, Sao-Jie; Lin, Guang-Huei; Hsiung, Pao-Ann ; Hu, Yu-Hen: *Hardware Software Co-Design of a Multimedia SOC Platform*. Springer, 2010. – ISBN 978–90–481–8171–1

[CM03] Corwin, Michael P.; Morris, Dale C.: *Method and implementation of statistical detection of read after write and write after write hazards*. April 15 2003. – US Patent 6,550,001

[CMS06] Claus, Christopher; Müller, Florian H. ; Stechele, Walter: Combitgen: A new approach for creating partial bitstreams in Virtex-II Pro devices. In: *Workshop on reconfigurable computing Proceedings (ARCS 06)*, 2006, S. 122–131

[Cob17] Cobham Group: *LEON3 Processor*. http://www.gaisler.com/index.php/products/processors/leon3. Version: Oktober 2017

[Die17] Diestel, Reinhard: *Graphentheorie*. Springer, 2017 (Springer Lehrbuch). – ISBN 978–3–96134–004–0

[DPF12] Däne, Bernd; Pacholik, Alexander ; Fengler, Wolfgang: Aspekte zur Realisierung eines FPGA-basierten Softcore-Prozessors für den Einsatz in Regelungs- und Messsystemen. In: *EKA 2012 Beschreibungsmittel, Methoden, Werkzeuge und Anwendungen* (2012)

[DPZ+13] Däne, Bernd; Pacholik, Alexander; Zschäck, Stephan; Fengler, Wolfgang; Ament, Christoph ; Braune, Tobias: Designing a Control Application by Using a Specialized Multi-Core Soft Microprocessor. In: *IFAC Proceedings Volumes* 46 (2013), Nr. 28, S. 221–226. http://dx.doi.org/10.3182/20130925-3-CZ-3023.00034. – DOI 10.3182/20130925–3–CZ–3023.00034. – ISSN 1474–6670

[DW15] Dröschel, Wolfgang; Wiemers, Manuela: *Das V-Modell 97: der Standard für die Entwicklung von IT-Systemen mit Anleitung für den Praxiseinsatz*. Walter de Gruyter, 2015. – ISBN 978–3–486–80026–5

[Ebr15] Ebrahim, Ali: Dynamic partial reconfiguration management for high performance and reliability in FPGAs. (2015)

[EF04] Evans, Eric; Fowler, Martin: *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004. – ISBN 978–0–321–12521–7

[EJH+10] Esko, Otto; Jaaskelainen, Pekka; Huerta, Pablo; Carlos, S; Takala, Jarmo ; Martinez, Jose I.: Customized Exposed Datapath Soft-Core Design Flow with Compiler Support. In: *International Conference on Field Programmable Logic and Applications* IEEE, 2010. – ISSN 1946–1488, S. 217–222

[Eul13] Eulenstein, Michael: *Generierung portabler Compiler: das portable System POCO*. Springer, 2013 (Informatik-Fachberichte). – ISBN 978–3–642–73431–1

[FKSH09] Friedrich, Jan; Kuhrmann, Marco; Sihling, Marc ; Hammerschall, Ulrike: *Das V-Modell XT*. Springer, 2009 (Informatik im Fokus). – ISBN 978–3–642–01488–8

[Gro11] Grout, Ian: *Digital systems design with FPGAs and CPLDs*. Elsevier, 2011. – ISBN 978–0–08–055850–9

[Her12] Hermann, Martin: *Numerische Mathematik*. Walter de Gruyter, 2012. – ISBN 978–3–486–71970–3

[HKT11] Hansen, Simen G.; Koch, Dirk ; Torresen, Jim: High speed partial run-time reconfiguration using enhanced ICAP hard macro. In: *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum* IEEE, 2011. – ISSN 1530–2075, S. 174–180

[HMN09] Hackenberg, Daniel; Molka, Daniel ; Nagel, Wolfgang E.: Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems. In: *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* IEEE, 2009. – ISSN 1072–4451, S. 413–422

[HPG+11] Hausotte, Tino; Percle, Brandon; Gerhardt, Uwe; Dontsov, Denys; Manske, Eberhard ; Jäger, Gerd: Homodyne interference signal demodulation for nanopositioning and nanomeasuring machines. In: *Innovation in Mechanical Engineering–Shaping the Future: proceedings; 56. IWK 2011, 12-16 September 2011, Ilmenau, University of Technology* Bd. 56, 2011

[IEE85] IEEE Computer Society Standards Committee. Working group of the Microprocessor Standards Subcommittee: IEEE standard 754 for binary floating-point arithmetic Institute of Electrical and Electronic Engineers, 1985

[IEE90] IEEE Standards Coordinating Committee and others: IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990). Los Alamitos. In: *CA: IEEE Computer Society* 169 (1990)

[Int18] Intel FPGA and SoC: *Partial Reconfiguration User Guide: Intel Quartus Prime Pro Edition*. `https://www.altera.com/documentation/tnc1513987819990.html`. Version: Mai 2018

[IS93] Iseli, Christian; Sanchez, Eduardo: Spyder: A reconfigurable VLIW processor using FPGAs. In: *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines* IEEE, 1993, S. 17–24

[Kar14] Karlin, Samuel: *A First Course in Stochastic Processes*. Elsevier, 2014. – ISBN 978–1–4832–6809–5

[KBJV06] Kurtev, Ivan; Bézivin, Jean; Jouault, Frédéric ; Valduriez, Patrick: Model-based DSL frameworks. In: *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications* ACM, 2006 (OOPSLA '06). – ISBN 1–59593–491–X, S. 602–616

[Kem11] Kemnitz, Günter: *Technische Informatik: Band 2: Entwurf digitaler Schaltungen*. Springer, 2011 (eXamen.press). – ISBN 978–3–642–17447–6

[KKSF17] Kirchhoff, Michael; Kaptsova, Natalia; Streitpferdt, Detlef ; Fengler, Wolfgang: Optimizing compiler for a specialized real-time floating point softcore processor. In: *8th Annual Industrial Automation and Electromechanical Engineering Conference (IEMECON)* IEEE, 2017, S. 181–188

[Koc14] Koch, Dirk: *Partial Reconfiguration on FPGAs: Architectures, Tools and Applications*. Springer, 2014 (Lecture Notes in Electrical Engineering). – ISBN 978–1–4899–9356–4

[KR07] Kuon, I.; Rose, J.: Measuring the Gap Between FPGAs and ASICs. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26 (2007), Feb, Nr. 2, S. 203–215. http://dx.doi.org/10.1109/TCAD.2006.884574. – DOI 10.1109/TCAD.2006.884574. – ISSN 0278–0070

[Kre18] Kreuzkamp, Björn-Philipp: *Konzeption und Realisierung eines Benchmark-Tools für einen optimierenden Compiler*, Technische Universität Ilmenau, Bachelorarbeit, 2018

[KS97] Krishna, C.M.; Shin, K.G.: *Real-time Systems*. McGraw-Hill, 1997 (Computer engineering). – ISBN 0–07–057043–4

[KT08] Kelly, Steven; Tolvanen, Juha-Pekka: *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, 2008 (Wiley - IEEE). – ISBN 978–0–470–24925–3

[KTR08] Kuon, Ian; Tessier, Russell ; Rose, Jonathan: FPGA architecture: Survey and challenges. In: *Foundations and Trends in Electronic Design Automation* 2 (2008), Nr. 2, S. 135–253. ISBN 978–1–60198–126–4

[Kun73] Kunsemüller, Horst: Assembler und Compiler. In: *Betriebsprogramme in Rechenanlagen*. Springer, 1973. – ISBN 978–3–519–06514–2, S. 157–185

[KWS+18] Kirchhoff, Michael; Weisensee, Jörn; Streitferdt, Detlef; Fengler, Wolfgang ; Rozova, Elena: Increasing Efficiency in Data Flow Oriented Model Driven Software Development for Softcore Processors. In: *IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)* Bd. 02 IEEE, 2018. – ISSN 0730–3157, S. 806–811

[LD09] Lai, Victor; Diessel, Oliver: ICAP-I: A reusable interface for the internal reconfiguration of Xilinx FPGAs. In: *International Conference on Field-Programmable Technology*, 2009. – ISBN 978–1–4244–4375–8, S. 357–360

[LL13] Ludewig, Jochen; Lichter, Horst: *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken.* dpunkt.verlag, 2013. – ISBN 978–3–86491–299–3

[LTT11] Lau, Kung-Kiu; Taweel, Faris M. ; Tran, Cuong M.: The W Model for Component-Based Software Development. In: *37th EUROMICRO Conference on Software Engineering and Advanced Applications* IEEE, 2011. – ISSN 1089–6503, S. 47–50

[Mar17] Marwedel, Peter: *Embedded system design.* Springer, 2017 (Embedded Systems). – ISBN 978–3–319–56045–8

[Mat01] Mathew, Binu: *Very Large Instruction Word Architectures (VLIW Processors and Trace Scheduling).* CRC Press, 2001. – 5–10 S. – ISBN 978–0–8493–0885–7

[McI68] McIlroy, M. D.: Mass-Produced Software Components. In: Buxton, J. M. (Hrsg.); Naur, Peter (Hrsg.) ; Randell, Brian (Hrsg.): *Software Engineering Concepts and Techniques (1968 NATO Conference of Software Engineering)*, NATO Science Committee, Oktober 1968, S. 88–98

[Mei10] Meisel, André: *Design Flow für IP basierte, dynamisch rekonfigurierbare, eingebettete Systeme.* Univ.-Verlag (TU Chemnitz), 2010. – ISBN 978–3–941003–15–6

[MKW11] Matalon, Shabtay; Klein, Russell ; Walls, Colin: Embedded System Power Consumption: A Software or Hardware Issue? / Mentor Graphics. 2011. – Forschungsbericht

[MMF14] Müller, Marcus; Machleidt, Torsten ; Fengler, Wolfgang: SoC Design for Complex Standalone Optical Measurement Devices. In: *Autonomous Systems 2014 Proceedings of the 7 th GI Workshop*, 2014, S. 66–75

[Moo98] Moore, Gordon E.: Cramming more components onto integrated circuits. In: *Proceedings of the IEEE* 86 (1998), Nr. 1, S. 82–85. `http://dx.doi.org/10.1109/JPROC.1998.658762`. – DOI 10.1109/JPROC.1998.658762. – ISSN 0018–9219

[Mül14] Müller, Marcus: Beitrag zum modellbasierten Entwurf eingebetteter Systeme: komponentenbasierte Modellierungsansätze für verteilte rekonfigurierbare Plattformen ISLE, 2014

[Nat12] National Instruments: *FPGA Fundamentals.* `http://www.ni.com/white-paper/6983/en/`. Version: Mai 2012

[New08] Newcomb, Jamie D.: *A scalable approach to multi-core prototyping*, Virginia Tech, Diss., 2008

[PCCC05] Pankratiev, E. V.; Chepovskiy, A. M.; Cherepanov, E. A. ; Chernyshev, S. V.: Algorithms and Methods for Solving Scheduling Problems and Other Extremum Problems on Large-Scale Graphs. In: *Journal of Mathematical Sciences* 128 (2005), Nr. 6, S. 3487–3495. `http://dx.doi.org/10.1007/s10958-005-0283-z`. – DOI 10.1007/s10958–005–0283–z. – ISSN 1573–8795

[PH17] Patterson, David A.; Hennessy, John L.: *Computer Architecture: A Quantitative Approach*. Elsevier, 2017 (The Morgan Kaufmann Series in Computer Architecture and Design). – ISBN 978–0–12–811905–1

[Pin08] Pinedo, Michael L.: *Scheduling: theory, algorithms, and systems*. Springer, 2008. – ISBN 978–3–319–26580–3

[PKMF11] Pacholik, A; Klöckner, J; Müller, M ; Fengler, W: LiSARD: programmierbarer Rechenkern für rechenintensive Echtzeitdatenverarbeitung mit PXI der R-Serie. In: *Virtuelle Instrum Praxis* (2011), S. 217–21

[PP85] Page, D.W.; Peterson, L.V.R.: *Re-programmable PLA*. `http://www.google.com/patents/US4508977`. Version: April 2 1985. – US Patent 4,508,977

[PR84] Puttkamer, Ewald von; Rissberger, Alfons: Übersetzung und Interpretation von Sprachen. In: *Informatik für technische Berufe*. Vieweg+Teubner Verlag, 1984 (MikroComputer-Praxis). – ISBN 978–3–519–02524–5, S. 139–154

[RA14] Rodriguez, Jaime C.; Ackermann, Kurt F.: Leveraging partial dynamic reconfiguration on zynq soc fpgas. In: *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2014 9th International Symposium on* IEEE, 2014, S. 1–6

[RFG16] Rettkowski, Jens; Friesen, Konstantin ; Göhringer, Diana: RePaBit: Automated generation of relocatable partial bitstreams for Xilinx Zynq FPGAs. In: *International Conference on ReConFigurable Computing and FPGAs (ReConFig)* IEEE, 2016, S. 1–8

[Roy87] Royce, Winston W.: Managing the Development of Large Software Systems: Concepts and Techniques. In: *Proceedings of the 9th International Conference on Software Engineering* IEEE Computer Society Press, 1987 (ICSE '87). – ISBN 0–89791–216–0, S. 328–338

[Rze94] Rzehak, Helmut: *Die Echtzeitdatenverarbeitung: Grundlagen und Methoden für die Praxis*. Wiesbaden : Vieweg+Teubner Verlag, 1994. – 13–41 S. `http://dx.doi.org/10.1007/978-3-322-85506-0_2`. `http://dx.doi.org/10.1007/978-3-322-85506-0_2`. – ISBN 978–3–322–85506–0

[SEHV12] Stahl, Thomas; Efftinge, Sven; Haase, Arno ; Völter, Markus: *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. dpunkt.verlag, 2012. – ISBN 978–3–89864–881–3

[Sem09] Semiconductor Industry Association: *ITRS: International technology roadmap for semiconductors*. 2009

[SKV+06] Sheldon, David; Kumar, Rakesh; Vahid, Frank; Tullsen, Dean ; Lysecky, Roman: Conjoining soft-core FPGA processors. In: *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design* ACM, 2006 (ICCAD '06). – ISBN 1–59593–389–1, S. 694–701

[Sta17] Standard Performance Evaluation Corporation: *Standard Performance Evaluation Corporation Benchmarks*. `https://www.spec.org/benchmarks.html`. Version: 2017

[Sti16] Stiny, Leonhard: *Anwendungsspezifische Integrierte Bausteine*. Springer, 2016. – 621–670 S. http://dx.doi.org/10.1007/978-3-658-09153-8_11. http://dx.doi.org/10.1007/978-3-658-09153-8_11. – ISBN 978–3–658–14387–9

[SVM01] Sangiovanni-Vincentelli, A.; Martin, G.: Platform-based design and software design methodology for embedded systems. In: *IEEE Design Test of Computers* 18 (2001), Nov, Nr. 6, S. 23–33. http://dx.doi.org/10.1109/54.970421. – DOI 10.1109/54.970421. – ISSN 0740–7475

[SWM+06] Saleh, Resve; Wilton, Steve; Mirabbasi, Shahriar; Hu, Alan; Greenstreet, Mark; Lemieux, Guy; Pande, Partha P.; Grecu, Cristian ; Ivanov, Andre: System-on-chip: Reuse and Integration. In: *Proceedings of the IEEE* 94 (2006), Nr. 6, S. 1050–1069. http://dx.doi.org/10.1109/JPROC.2006.873611. – DOI 10.1109/JPROC.2006.873611. – ISSN 0018–9219

[SWS+11] Steiner, Neil; Wood, Aaron; Shojaei, Hamid; Couch, Jacob; Athanas, Peter ; French, Matthew: Torc: towards an open-source tool flow. In: *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays* ACM, 2011. – ISBN 978–1–4503–0554–9, S. 41–44

[TC12] Thammasan, Nattapong; Chongstitvatana, Prabhas: Design of a GPU-styled softcore on field programmable gate array. In: *Ninth International Conference on Computer Science and Software Engineering (JCSSE)* IEEE, 2012, S. 142–146

[TEKV14] Tarrillo, Jimmy; Escobar, Fernando A.; Kastensmidt, Fernanda L. ; Valderrama, Carlos: Dynamic partial reconfiguration manager. In: *IEEE 5th Latin American Symposium on Circuits and Systems* IEEE, 2014, S. 1–4

[TG06] Tanenbaum, Andrew S.; Goodman, James: *Informatik : Rechnerarchitektur*. Bd. 2: *Computerarchitektur: Strukturen, Konzepte, Grundlagen*. Pearson Studium, 2006. – ISBN 978–3–8273–7151–5

[TPB+07] Trompeter, Jens; Pietrek, Georg; Beltran, Juan Carlos F.; Holzer, Boris; Kamann, Thorsten; Kloss, Michael; Mork, Steffen A.; Niehues, Benedikt ; Thoms, Karsten: *Modellgetriebene Softwareentwicklung: MDA und MDSD in der Praxis*. 2007. – ISBN 978–3–939084–11–2

[Tre17] Trenz Electronic GmbH: *TE0720 TRM*. http://www.trenz-electronic.de/fileadmin/docs/Trenz_Electronic/Modules_and_Module_Carriers/4x5/TE0720/REV03/Documents/TRM-TE0720-03.pdf. Version: November 2017

[Tre18a] Trenz Electronic GmbH: *TE0703-04 Digital Picture (Top View)*. http://www.trenz-electronic.de/fileadmin/docs/Trenz_Electronic/Modules_and_Module_Carriers/4x5/4x5_Carriers/TE0703/REV04/Pictures/TE0703-04%20top.jpg. Version: 2018

[Tre18b] Trenz Electronic GmbH: *TE0720-02 Digital Picture (Top View)*. http://www.trenz-electronic.de/fileadmin/docs/Trenz_Electronic/Modules_and_Module_Carriers/4x5/TE0720/REV02/Pictures/TE0720-02-2IFC3%20top.jpg. Version: 2018

[Ung01] Ungerer, Theo: Mikroprozessoren Stand der Technik und Forschungstrends. In: *Informatik-Spektrum* 24 (2001), Nr. 1, S. 3–15. http://dx.doi.org/10.1007/s002870000139. – DOI 10.1007/s002870000139. – ISSN 0170–6012

[VB02] Vandewoude, Yves; Berbers, Yolande: An overview and assessment of dynamic update methods for component-oriented embedded systems. In: *Proceedings of The International Conference on Software Engineering Research and Practice*, 2002, S. 521–527

[Ver00] Versteegen, Gerhard: *Das V-Modell in der Praxis: Grundlagen, Erfahrungen, Werkzeuge*. dpunkt.verlag, 2000. – ISBN 3–932588–39–8

[VF14] Vipin, Kizheppatt; Fahmy, Suhaib A.: ZyCAP: Efficient partial reconfiguration management on the Xilinx Zynq. In: *IEEE Embedded Systems Letters* 6 (2014), Nr. 3, S. 41–44. http://dx.doi.org/10.1109/LES.2014.2314390. – DOI 10.1109/LES.2014.2314390. – ISSN 1943–0663

[VG06] Vahid, Frank; Givargis, Tony: *Embedded System Design: A Unified Hardware/-Software Introduction*. Wiley, 2006. – ISBN 978–81–265–0837–2

[WAB08] Wong, S.; As, T. van ; Brown, G.: p-VEX: A reconfigurable and extensible softcore VLIW processor. In: *International Conference on Field-Programmable Technology*, 2008, S. 369–372

[Wag17] Wagner, Lothar: *Methodische Konzeption und praktische Realisierung einer Erweiterung eines Single-Core Softprozessors zu einer Multi-Core Architektur*, Technische Universität Ilmenau, Masterarbeit, 2017

[Wal74] Waldschmidt, Helmut: *Optimierungsfragen im Compilerbau*. Hanser, 1974. – ISBN 3–446–11895–0

[Wec15] Wecker, Dieter: *Prozessorentwurf: Von der Planung bis zum Prototyp*. Walter de Gruyter, 2015. – ISBN 978–3–11–040296–4

[Xil10] Xilinx Inc.: *Xilinx Design Reuse Methodology for ASIC and FPGA Designers*. 2010

[Xil13] Xilinx Inc.: *Vivado Design Suite 7 Series FPGA and Zynq-7000 All Programmable SoC Libraries Guide*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_3/ug953-vivado-7series-libraries.pdf#page=72&zoom=auto,-205,503. Version: Oktober 2013

[Xil14a] Xilinx Inc.: *Virtex-II Platform FPGAs: Complete Data Sheet*. https://www.xilinx.com/support/documentation/data_sheets/ds031.pdf. Version: April 2014

[Xil14b] Xilinx Inc.: *Vivado Design Suite User Guide Design Flows Overview*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug892-vivado-design-flows-overview.pdf. Version: April 2014

[Xil15] Xilinx Inc.: *Vivado Design Suite User GuidePartial Reconfiguration*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/ug909-vivado-partial-reconfiguration.pdf. Version: Juni 2015

[Xil16a] Xilinx Inc.: *7 Series FPGAs Configurable Logic Block User Guide*. https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf. Version: September 2016

[Xil16b] Xilinx Inc.: *AXI HWICAP v3.0*. https://www.xilinx.com/support/documentation/ip_documentation/axi_hwicap/v3_0/pg134-axi-hwicap.pdf. Version: Oktober 2016

[Xil17a] Xilinx Inc.: *CORDIC v6.0 LogiCORE IP Product Guide*. https://www.xilinx.com/support/documentation/ip_documentation/cordic/v6_0/pg105-cordic.pdf. Version: Dezember 2017

[Xil17b] Xilinx Inc.: *MicroBlaze Micro Controller System v3.0*. https://www.xilinx.com/support/documentation/ip_documentation/microblaze_mcs/v3_0/pg116-microblaze-mcs.pdf. Version: Dezember 2017

[Xil17c] Xilinx Inc.: *Vivado Design Suite User Guide - Partial Reconfiguration*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug909-vivado-partial-reconfiguration.pdf. Version: Juni 2017

[Xil18a] Xilinx Inc.: *7 Series FPGAs Configuration User Guide*. https://www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf. Version: August 2018

[Xil18b] Xilinx Inc.: *Partial Reconfiguration Controller v1.3*. https://www.xilinx.com/support/documentation/ip_documentation/prc/v1_3/pg193-partial-reconfiguration-controller.pdf. Version: April 2018

[Xil18c] Xilinx Inc.: *UltraScale Architecture and Product Data Sheet: Overview*. https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf. Version: November 2018

[Xil18d] Xilinx Inc.: *UltraScale Architecture Configuration User Guide*. https://www.xilinx.com/support/documentation/user_guides/ug570-ultrascale-configuration.pdf. Version: August 2018

[Xil18e] Xilinx Inc.: *Vivado Design Suite User Guide: Partial Reconfiguration*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug909-vivado-partial-reconfiguration.pdf. Version: Juni 2018

[Xil18f] Xilinx Inc.: *Zynq-7000 SoC Data Sheet: Overview*. https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf. Version: Juli 2018

[Xil18g] Xilinx Inc.: *Zynq-7000 SoC Technical Reference Manual*. https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf. Version: Juli 2018

[Xil18h] Xilinx Inc.: *Zynq-7000 SoC (Z-7007S, Z-7012S, Z-7014S, Z-7010, Z-7015, and Z-7020): DC and AC Switching Characteristics*. https://www.xilinx.com/support/documentation/data_sheets/ds187-XC7Z010-XC7Z020-Data-Sheet.pdf. Version: Juli 2018

[YI72] Yoshida, Kenji; Ichikawa, Osamu: *Method of manufacturing semiconductor integrated circuits*. `http://www.google.com/patents/US3634929`. Version: Januar 18 1972. – US Patent 3,634,929

[YYX+12] Yu, Zhiyi; You, Kaidi; Xiao, Ruijin; Quan, Heng; Ou, Peng; Ying, Yan; Yang, Haofan; Zeng, Xiaoyang u. a.: An 800MHz 320mW 16-core processor with message-passing and shared-memory inter-core communication mechanisms. In: *IEEE International Solid-State Circuits Conference* IEEE, 2012. – ISSN 2376–8606, S. 64–66

[ZAM+10] Zschäck, Stephan; Amthor, Arvid; Müller, Marcus; Klöckner, Johannes; Ament, Ch ; Fengler, Wolfgang: Integrated system development process for high-precision motion control systems. In: *IEEE International Conference on Control Applications* IEEE, 2010. – ISSN 1085–1992, S. 344–350

# Erklärung

Ich versichere, dass ich die vorliegende Arbeit ohne unzulüssige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Weitere Personen waren an der inhaltlich-materiellen Erstellung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich hierfür nicht die entgeltliche Hilfe von Vermittlungs- bzw. Beratungsdiensten (Promotionsberater oder anderer Personen) in Anspruch genommen. Niemand hat von mir unmittelbar oder mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer Prüfungsbehörde vorgelegt.

Ich bin darauf hingewiesen worden, dass die Unrichtigkeit der vorstehenden Erklärung als Täuschungsversuch bewertet wird und gemäß Paragraph 7 Absatz 10 der Promotionsordnung den Abbruch des Promotionsverfahrens zur Folge hat.

Ilmenau, den 10.01.2019 Michael Kirchhoff